Launchpad: A Rhythm-Based Level Generator for 2-D Platformers

Gillian Smith, *Student Member, IEEE*, Jim Whitehead, *Senior Member, IEEE*, Michael Mateas, Mike Treanor, Jameka March, and Mee Cha

***Abstract*—Launchpad is an autonomous level generator that is based on a formal model of 2-D platformer level design. Levels are built out of small segments called "rhythm groups," which are generated using a two-tiered, grammar-based approach. These segments are pieced together into complete levels that are then rated according to a set of design heuristics. Generation can be controlled using a set of parameters that influence the level pacing and geometry. The approach minimizes the amount of content that must be manually authored: instead of piecing together large segments of a level, Launchpad uses base components that are commonly found in a number of 2-D platformers. Launchpad produces an impressive variety of levels which are all guaranteed to be playable.**

***Index Terms*—Artificial intelligence, games, level design, multilevel grammars, procedural content generation.**

I. Introduction

A GAME level acts as a "container for gameplay" [1], providing the player with a space to explore and master the game's mechanics. Good level design is vital to a game's success. Despite this importance, the science behind level design is poorly understood. With the rising interest in procedural level generation, both as a research area and as a solution to content creation costs, it is important for us to formally model how to recognize and design good levels. This deep understanding helps us move beyond level generators that work by stitching together larger chunks of human-authored content, and are instead capable of creating interesting levels from basic level components. This paper presents Launchpad, a level generator for 2-D platformers that is based on a model of rhythmic player movement; this model is derived from analyzing existing games in the genre.

Two-dimensional platformers (such as *Super Mario World* [2] and *Sonic the Hedgehog* [3]) are well suited to research in procedural level generation. Games in this genre have simple rules but exhibit emergently complex level design. Furthermore, game levels have a heavy influence on the player's experience in the game. This allows us to focus specifically on level design issues such as physics constraints and challenge structure without facing complications from more sophisticated game mechanics or narrative elements.

A key underlying idea for 2-D platformer level design, in particular those that focus on dexterity-based challenges, is a notion of rhythm and the timing and repetition of distinct user actions [4]–[6]. Players strive to navigate complicated playfields full of obstacles and collectible items. Manually designed levels frequently contain a series of challenging jumps that must be perfectly timed. The level generator presented in this paper, called Launchpad, realizes this theory of level design. In order to capture the importance of rhythm, the level generator is designed with a two-tiered, grammar-based approach, where the first tier is a rhythm generator and the second tier creates geometry based on that rhythm; the end result of this process is a set of "rhythm groups" that can be combined to form complete levels. This separation of tiers ensures that the intended rhythm is always present, regardless of geometric representation.

Launchpad allows a designer to refine its generative space by manipulating parameters that have a clear and intuitive impact on generated levels. These parameters dictate a general path that the level should take, the types and frequencies of geometry components, and how collectible items are distributed throughout the level. Adjusting these parameters can drastically alter the generated level.

The wide range of levels that Launchpad can generate points to a greater need for techniques to evaluate the expressive range of content creators. A common strategy for evaluating a generator is to show characteristic examples of the kinds of content that can be produced alongside statistics on how many levels can be produced and how quickly. While this approach provides useful and interesting information about the generator, it does not fully capture the range of content that can be created, nor does it easily support analysis of how this range changes for different fitness functions and generation parameters. We call this quality of a content generator "expressive range."

Off-the-shelf generators exist for procedural animations [7], cities [8], and natural features [9]. As more of these tools become available, it becomes increasingly important for a potential user to be able to understand the kinds of content a generator will give them in order to compare potential generation techniques. Furthermore, understanding the expressive range of a content generator is useful in driving future work in procedural content generation because it can uncover unexpected biases and dependencies in the generator. This paper also presents techniques for visualizing the expressive range of Launchpad through applying external heuristics and clustering generated content to easily explore the generated space.

Manuscript received September 09, 2010; revised November 15, 2010; accepted November 17, 2010. Date of publication November 29, 2010; date of current version March 16, 2011.

G. Smith, J. Whitehead, M. Mateas, M. Treanor, and J. March are with the Expressive Intelligence Studio, University of California Santa Cruz, Santa Cruz, CA 95064 USA (e-mail: gsmith@soe.ucsc.edu).

M. Cha was with the Expressive Intelligence Studio, University of California, Santa Cruz, Santa Cruz, CA 95064 USA. She is now with OnLive, Palo Alto, CA 94031 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2010.2095855

The primary contributions of this work are:

1) a formal model of the components and structure of platformer levels;
2) an autonomous, designer-guided level generator for 2-D platform games based on this model;
3) methods for evaluating and visualizing the expressivity of a content generator, as applied to Launchpad.

II. Related Work

A. Domain Analysis for Level Design

There are a few books on game design that address concepts relevant to level design. For example, Salen and Zimmerman's *Rules of Play* is primarily about game design, but does discuss important level design concepts such as repetition and interactivity when discussing crafting the play of experience [10]. Books that do address level design primarily address its general principles. *Level Design for Games* [11] discusses the importance of spatial layout and atmosphere. *Fundamentals of Game Design* [12] and *Beginning Game Level Design* [13] also largely focus on general issues, giving a couple of paragraphs on level design for specific genres. Short descriptions of genre-specific level design are not sufficient to provide a detailed understanding of the structural interrelationships of elements in a level.

While there has not been a great deal of analysis for genre-specific level design, there are a few notable exceptions. Nelson's article on level elements in *Breakout* is a good example of the kind of detail needed to fully model levels [14]. His article takes a reductionist view of level design for *Breakout*-style games, first decomposing levels into their genre-specific constituent components, and then giving rules for how to compose level elements into interesting designs. This approach of reducing levels to their base components and giving rules for their recombination is the essence of the modeling technique we used in analyzing platformer levels. There is also work in identifying and analyzing design patterns in levels. Hullett and Whitehead identify a number of common idioms used in first person shooter (FPS) levels [15], such as arenas and sniper nests. While difficult to compare analyses of such different genres, these idioms seem analogous to the cells in our model for platformer levels, as each idiom contains a number of different, potentially disjointed player actions. Milam and El Nasr identify patterns in levels covering a number of different genres [16]; however, these patterns are more focused on categorizing player behavior and motivation, rather than level geometry. In future work, patterns such as these may be useful for guiding level generation based on our framework presented here.

There are also two papers analyzing level design aspects of platformers in particular. Boutros writes about common design goals in the bestselling platform games [17], focusing especially on visuals, controls, rewards, and challenges. He analyzes the first 5–10 min of gameplay for a number of games, listing the challenges that the player will face. Dormans writes about embodiment, flow, and discovery in platformer levels, and provides numerous examples of common tropes in platformer levels [18]. Missing in these two papers is a formal model of level design and challenge structure, a prerequisite for a level generator that cannot rely on hand-authored level components.

The importance and structure of challenge is central to our model for how level components fit together, and especially in our notion of rhythm groups. In his book describing the role of a level designer, Byrne claims that challenge is the most important aspect of level design [1] because it is the key to the player enjoying the level. Nicollet also discusses the role of challenge in dexterity-based games [6]. He creates a series of rules for designing challenge, including the importance of timing. These rules have provided insight on how to segment levels into rhythm groups.

B. Procedural Content Generation

Procedural content generation is commonly used in games to improve replayability by providing a new scenario each time the player opens the game, or modifying the scenario whenever the player makes a new choice. Recently it has also been used at design time to help a designer create their own content for a specific game. This section discusses techniques in procedural content generation and how a player or a designer can interact with a generator.

The first major game with procedural content generation was *Rogue* [19], an ASCII role-playing game (RPG) created in 1980. Its levels consist of arbitrarily sized rooms connected by corridors, with enemies and treasure placed throughout them. Enemies get progressively harder to defeat as play progresses, and different treasure is scattered around the world: each time the player loads the game, she is faced with a different dungeon to explore. *Rogue*'s popularity spawned a number of "roguelikes," both ASCII-based and graphical. *Diablo* [20] was one of the first graphical games with procedural content: much like in *Rogue*, levels are still procedurally generated, but the content placed into the levels such as wall textures, pillars, and enemies are hand-authored. Roguelike approaches to level generation [21] tend to be "bottom-up" techniques that involve fitting together relatively large chunks of a level together according to simple constraints. Roguelikes typically do not allow a user to exert much control over the kinds of levels that are produced.

Spelunky [22] is an example of a recent roguelike-platformer hybrid: *Spelunky* is a cave-exploring 2-D platformer game, rather than the typical top-down RPG, which builds its levels out of static, arbitrarily sized hand-authored pieces. These pieces may contain many different actions for the player to take, as opposed to Launchpad's generator where geometry is independently generated for each action. The pieces are fit together on a square grid, with few rules governing how they should be placed. Though this results in random, uncontrolled level designs, this works because the player can modify the level using bombs, or navigate seemingly impossible areas with other tools such as ropes. Furthermore, the player is not expected to be able to reach everything in a given level. This generation technique is quite specific to the particular game design, but would be inappropriate for, say, a *Super Mario World*-style level due to the low level of authorial control it provides. Launchpad's generation technique is more broadly applicable; it is designed for games that favor dexterity-based challenges over exploration, and the underlying representation

is based on an analysis of a number of different platformer games.

A detailed level analysis is also the basis of Dormans's work in procedural level generation for action-adventure games [23]. Dormans examined the mission and level structure of *Legend of Zelda: Twilight Princess* levels. He also takes a two-layer, grammar-based approach to level generation, first producing missions using a grammar, then using shape grammars [24] to create playable levels from generated missions. These levels contain enemies, collectibles, and simple lock-and-key puzzles. In both his generator and Launchpad, the more abstract tier is used to provide a nongeometric structure based on intended player actions. This idea is well expressed by Ashmore and Nitsche [25] who, in discussing the lock-and-key quests they generate for *Charbitat*, state that "procedurally determined context is necessary to structure and make sense of this [procedurally generated] content."

Work that specifically addresses procedural level generation for 2-D platformers is that of Compton and Mateas [5]. They also split up levels into smaller sections and generate those sections from individual platform tilesets. Our work differs because our two-tiered approach to rhythm group generation, by explicitly separating rhythm generation and geometry generation, means we can represent more variety in levels than their pattern-building approach, in which rhythm is implicitly determined from geometry decisions. We also provide control over the output of the generation through a set of parameters that a user can manipulate. Mawhorter and Mateas [26] use a technique they call occupancy-regulated extension to generate Mario-style platformer levels. This works by defining rules for how hand-authored sections of a level fit together, and expanding levels from a single seed section. These sections can overlap with each other and form levels with multiple paths through them. Their approach creates highly varied levels with multiple paths and emergent properties, although at the expense of being able to guarantee playability. It also has a high authorial burden because the generator has no built-in understanding of the actions the player is taking: the generator is only as strong as the human-authored chunks it fits together. Launchpad requires very little human authoring effort and can guarantee that all levels it creates are playable, but currently cannot create levels with multiple paths.

Hullett and Mateas [27] provide authorial control over the scenarios that they generate for a collapsed structure emergency rescue training game. A designer can specify goals for the scenario, such as a half-flooded house or a room with collapsed walls. These goals are the input to a hierarchical task network (HTN) planner that performs the steps necessary to convert a sound structure into a partially collapsed structure with the desired qualities. Their approach differs from ours in that they start with an existing structure that is changed over time by the generator according to certain rules, rather than creating a structure from small components.

A separate strand of work in procedural content generation uses PCG techniques to personalize content in a game, either through adapting content to a particular skill level in real time, or by determining a model of player behavior at design time and creating a static level to fit that model. Launchpad instead acts on an implied model of ideal player "speedrun" behavior in which the player strives to be moving at maximum speed for as long as possible and perfectly times jumps to overcome obstacles. Work in this area includes personalized race tracks for racing games [28], evolving weapons in a space shooting game [29], and adapting Mario levels by manipulating parameters such as gap width and frequency [30]. The rhythm-based generation technique behind Launchpad has also been used in *Polymorph* [31], a platformer game with level generation-based dynamic difficulty adjustment.

Although not in the realm of content generation for games, other grammar-based approaches are related to our work. The Instant Architecture project [32] uses grammars both for architecture generation and ensuring the correct distribution of attributes. Tableau Machine [33], an art generator, uses a grammar for generating art, and their method for solving the problem of under-constrained results through generate and test was the inspiration for our own implementation of critics.

III. Deconstructing Platformers

In order to build a content generator, it is necessary to first have a formal model for that content. This model should describe both the basic components the generator can use as well as rules for how these components can be combined. We built our model for 2-D platformer levels by examining a number of games that are exemplary of the genre, including *Super Mario World* [2], the *Sonic the Hedgehog* series [3], [34], *Yoshi's Island DS* [35], and *Donkey Kong Country 2: Diddy's Kong Quest* [36]. These games reveal a number of different styles of platformer; for example, *Super Mario World* is characterized by a reward structure closely tied to the probability of failure, and a single path through the level with relatively few hidden areas accessible from vines or pipes. The focus of the level design is the challenge of completing each level through mastery of its dexterity challenges.

In contrast, *Sonic the Hedgehog* levels tend to have multiple paths to completion with rings liberally spread throughout the level, making it easy to collect rewards: a skilled player will collect a large number of rings and extra lives to carry throughout the game. The primary challenge in this game comes from mastery of dexterity challenges, but also from choosing an appropriate path through the level. Sonic's speed and agility, combined with this level structure, invites a "speed run" play style.

Donkey Kong Country 2 places less emphasis on speed runs and more emphasis on secret areas the player must find to collect rewards and secret coins. Some of these areas are accessed through mastering dexterity challenges, and many are hidden from the player and must be discovered.

Despite the differences between these styles, there are a number of similarities in their component elements and how those components fit together to form a level. All of these games require some mastery of dexterity-based challenge, although the number and difficulty of these challenges may vary. Furthermore, all of these games incorporate a sense of rhythm in their pacing.

This section provides an analysis of existing, human-designed platformer levels. The avatar as it relates to level design is described first, followed by a categorization of the basic

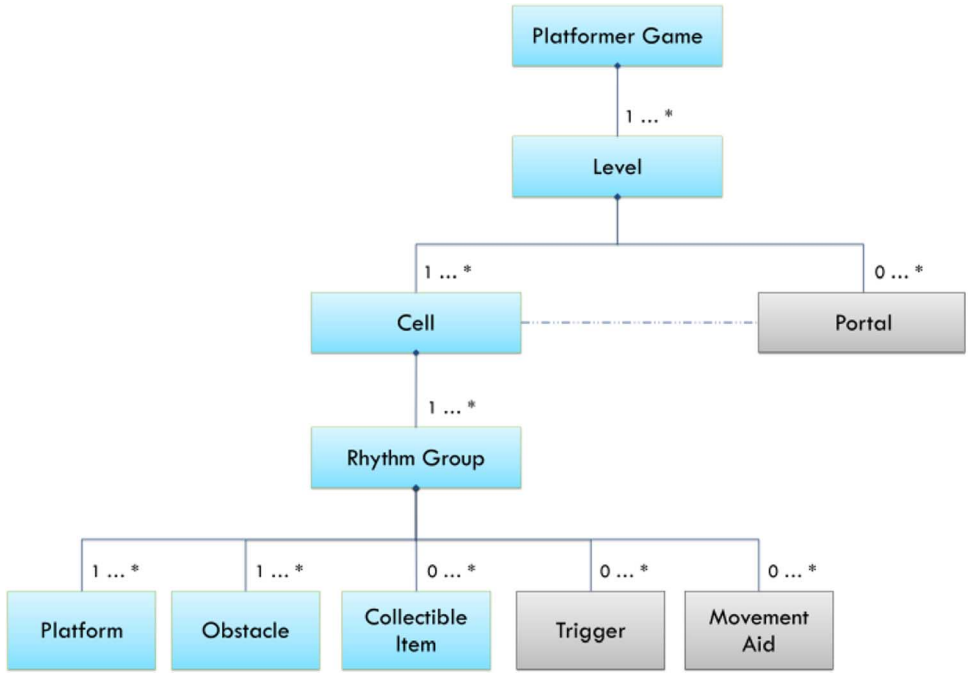

Fig. 1. Hierarchical level framework. This conceptual model shows how the components in a level fit in to a more abstract structure. The boxes highlighted in blue are implemented in Launchpad.

components found in levels, and finally a model for how these components fit together spatially. An overview of this framework is shown in Fig. 1.

A. The Avatar

The avatar is the character that is controlled by the player; although there are sometimes multiple playable characters, such as Diddy and Dixie Kong in *Donkey Kong Country 2*, the player can usually control only one character at any given time. Avatar choices are sometimes merely cosmetic, but there are often important differences in their behavior. For example, different babies in *Yoshi's Island DS* bestow different abilities on Yoshi and also change the physics of player movement, e.g., Baby Mario allows Yoshi to dash, Baby Peach allows Yoshi to glide.

An avatar's specific movement abilities vary per game, but can be approximated using a simple physics model. The player tends to have control over the avatar's horizontal movement, and limited control over vertical movement through jumping and crouching. The model for player physics is discussed in further detail in Section IV-B.

B. Level Components

Components of platformer levels are categorized by the roles they play in a level. It is possible for level components to be members of more than one category; for example, item boxes in *Super Mario World* are considered primarily as collectible items, but also as platforms because the avatar can walk along the top of them. Example level components and their properties include the following.

1) Platforms: A platform is defined as any surface that the avatar can walk or run across safely. Often, objects that serve some other primary purpose, such as item boxes in *Super Mario World*, double as platforms. Platforms have physical properties: a coefficient of friction, size, and slope. They can be in constant or occasional motion, and often form paths planned by the designer.

2) Obstacles: An obstacle is any construct in the level that is capable of imparting damage to the avatar or interrupts the flow of play by obstructing player movement. For example, a gap between two platforms is considered an *obstacle*, even though it is not explicitly represented by an object in the level. Other obstacles include moving enemies and spikes, as found in a number of platformers such as *Sonic the Hedgehog*. Obstacles can be overcome by either removing them from the level (e.g., by jumping on top of them), or by avoiding them entirely.

3) Movement Aids: Movement aids help the avatar navigate the level in some manner other than by the player's core movement mode. Examples of movement aids include ladders, ropes, springs, and moveable trampolines—all of these temporarily modify either the direction or speed of player movement. A spring is a movement aid that is used in Launchpad's level generator.

4) Collectible Items: All platform games have a reward system, almost always in the form of collectible items. Collectible items are any object in the level that provides a reward, such as coins, rings, power-ups, or weapons. The reward value often takes the form of points that can be redeemed in a variety of ways, such as extra lives in *Sonic the Hedgehog* or unlocked zones in *New Super Mario Bros.* [37]. Daniel Boutros gives a detailed account of the design and structure of reward systems in a variety of platform games [17].

Collectible items have a great deal of impact on the player's satisfaction in playing the game [38], but are also frequently used in platformers to guide the player through the level. For example, coins are often placed at the peak of jumps, or to indicate that the player should move in a certain direction (Fig. 2).

Fig. 2. Use of collectible items. In this segment from *Super Mario World*, Mario is poised to make a challenging set of jumps. The Yoshi coin has the highest reward, and is placed in the highest risk area: above a platform that can sink into the water. The three other coins show the player the ideal path from that sinking platform to safe ground. Finally, there is a collectible item in the form of an item box.

Fig. 3. Triggers. This section from a *Super Mario World* level shows an example of a trigger: the "p" button turns blocks into coins, allowing the player to collect a large reward at the expense of unleashing enemies inside.

5) Triggers: Triggers are interactive objects or gameplay actions that change the state of the level. For example, Fig. 3 shows an example of triggers from *Super Mario World*, where Mario can jump on the blue "P" button to turn all the blocks into coins. Some triggers are also timed; for example, *Yoshi's Island DS* requires Yoshi to jump on a red button that turns red platforms active. Yoshi then has a short amount of time to run across the platforms and continue the level.

Triggers can add an interesting puzzle element to a platform game. *Shift* [39] is a puzzle platformer that makes extensive use of two main types of trigger: the shift key itself, which flips the negative space in the level, and keys to unlock doors.

C. Rhythm as a Structural Representation

Rhythm and pacing are key to the player's enjoyment of a game [40], [41] and also contribute significantly towards the difficulty of platformers with dexterity-based challenges [6]. We therefore chose to use rhythm as a way to structurally represent levels as a combination of *rhythm groups*. Rhythm groups are short, nonoverlapping sets of components that encapsulate an area of challenge. Each rhythm group consists of a rhythm of player actions, and level geometry that corresponds to that rhythm.

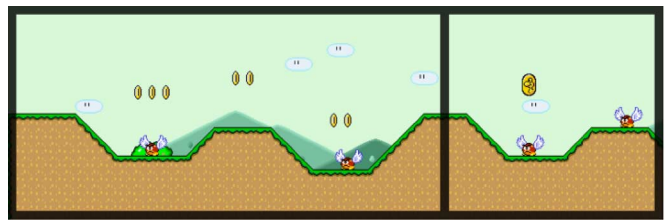

Fig. 4. Rhythm group example. This segment of a *Super Mario World* level shows two rhythm groups, each surrounded by a black rectangle. The rhythm groups are separated by a platform where no action is required to be taken, allowing the player a brief rest before moving on to deal with the next obstacle.

Rhythm can be found in platformers whenever the player performs actions such as jumping or shooting. These actions map directly to the controller, and the rhythm with which the ideal player must hit the buttons on the controller is the rhythm we are defining. For example, a rhythm could be a series of three short hops, or alternating jumping back and forth up a series of platforms that form a ladder.

Fig. 4 shows an example of two rhythm groups from *Super Mario World*. The leftmost group has the player perform two identical actions: jumping to collect coins and to kill flying goombas. Equally spaced between these two actions is the need to jump for two coins above the middle platform. The rightmost rhythm group is shorter version of the same rhythm and geometry pair. At first glance, it may appear as though the two groups shown should actually be a single group, since the platforms form a repeated pattern that the player must jump across. However, coin and enemy placement means that the components make up two separate groups. The key to this separation is the middle pair of coins in the first group, and their lack of a counterpart at the point where the two groups meet. In the first rhythm group, an ideal player would perform a series of jumps with minimal pauses to reach all the coins and kill the enemies. At the place where the groups meet, the player has a short opportunity to pause, allowing him to correctly time his jump to the next Yoshi coin and enemy. This pause forms a break in the rhythm that would not be there if the player had coins to jump for on that second peak.

D. Player Choice and Nonlinearity

Nonlinearity in platformer levels is represented by cells and portals. This nonlinearity manifests itself in a variety of ways. For example, in *Sonic the Hedgehog*, there are usually multiple paths through a level. There are moments when the player can switch between these paths, and the player often attempts to find the path that allows her to complete the level in the shortest amount of time. In our structure for levels, each point where the paths meet is a portal.

Cells define regions of nonoverlapping, linear gameplay. Their boundaries are set by the placement of transitions into and out of the regions, such as a secondary entrance to a level, a transition between paths through a level, or a portal to a secret area in the game. Fig. 5 shows two examples of cells and portals: the first is a scene from *Super Mario World* showing part of a cell with a portal marking an exit from it. The pipe in the top middle of the picture is a portal into a new area. The second example from *Sonic the Hedgehog* is more complex: the two

Fig. 5. Cells and portals. Two examples showing cells and portals. The upper figure shows a scene from *Super Mario World* with a single cell and a pipe that forms a portal out of it. The lower example is from *Sonic the Hedgehog 2* and shows three cells: the upper area, the lower area, and the secret coins area to the left. There are two portals between the upper and lower cells in the form of moving platforms between them. There is a single portal to the hidden coin area from the lower cell.

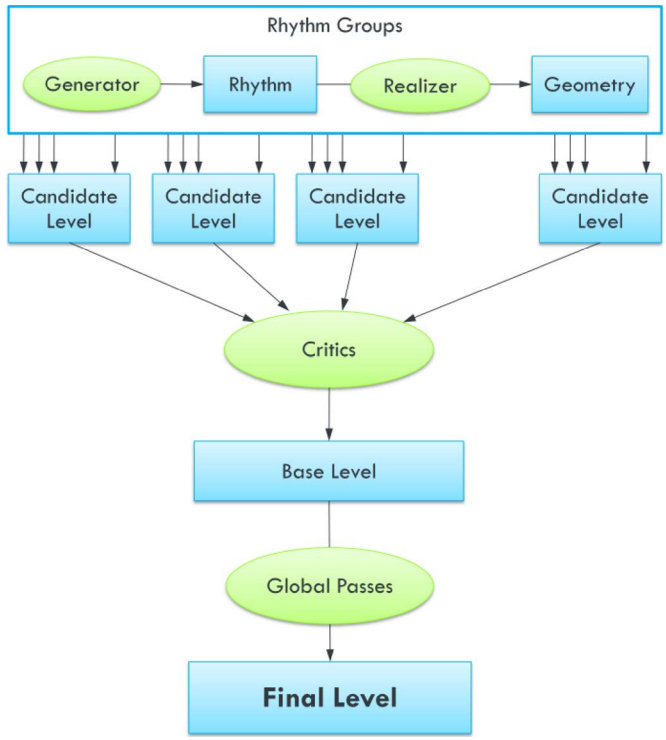

Fig. 6. Launchpad architecture diagram. Blue boxes denote artifacts produced by the system; green circles represent components of the generator, each of which is influenced by design parameters.

platforms marked as portals move up and down, providing a transition to a different path through the level. Note that for the sake of clarity, the figure does not have rhythm groups marked.

Knowing where the cells and portals are in a level helps us analyze their structure and catalog the many paths through a level. These paths may be of different difficulties and provide different rewards, depending on the rhythm groups that make up the cells along each path.

IV. Constructing Levels

Launchpad uses much of this level framework in automatically creating levels. To simplify the problem, we focus on generating rhythm groups and using them to build single-cell levels. Future work includes researching methods for incorporating triggers, secret areas, and multiple paths into a grammar-based level generator. Fig. 1 highlights the portions of the model for levels that are used in Launchpad. The platforms which Launchpad can create are flat or sloped, and can move along either horizontal or vertical paths. Implemented obstacles are enemies that should be killed, enemies that should be avoided (e.g., spikes), stompers, and gaps between platforms. Collectible items are placed during a global pass after the level has been generated: coins are positioned to guide players along jumps, and as decoration along long platforms.

The rhythm-based method for generating levels is described in Fig. 6. It begins with a two-layered, grammar-based approach for generating rhythm groups. The first stage creates a set of beats, where each beat corresponds to a player action. The second stage uses a grammar to realize this set of actions into corresponding geometry according to a set of physical constraints. This process creates a single rhythm group; many unique rhythm groups are used in creating levels.

To form a complete level, rhythm groups are fit together side by side, bridging them with a small platform that acts as a safe area for the player. Many different levels are generated, forming a pool of candidate levels which are then tested against a set of design heuristics that select the best level. This level is then modified by a global pass that adds collectible items according to rhythm and geometry information, and ties platforms to a common ground point.

Each stage of the level generation process has parameters that provide direct control over the output of the generator. Table I describes these parameters, and what stages they affect. A discussion of how these parameters affect the space of generated levels is in Section V-D.

A. Rhythm Generation

Rhythms have three main properties: type, length, and density. The rhythm generator first assigns a value to each of these properties based on the probabilities provided to the generator (Table I). Example timelines for different combinations of these

TABLE I
PARAMETERS FOR LAUNCHPAD THAT A USER CAN MANIPULATE

Rhythm	*Type*	A rhythm can be "regular", "swing", or "random". Regular rhythms have evenly spaced beats, swing rhythms have a short followed by long beat, and random rhythms have randomly spaced beats.
	Density	Rhythm density describes how closely beats are spaced. Density can be "low", "medium", or "high".
	Length	Rhythms can be 5, 10, 15, or 20 seconds long.
	Action Probabilities	The probability of a jump action occurring vs. a wait action occurring.
Geometry	*Jump Components*	The desired frequency for specific geometry being used in a level for a jump action. Geometry available: jumping up or down, with or without gaps; enemy; spring; fall.
	Wait Components	The desired frequency for specific geometry being used in the level for a wait action. Geometry available: stomper, moving platform.
	Repeating Geometry	The probability that a rhythm group should be immediately repeated.
Line	*Equation*	The path that the user would like the final level to follow, defined as a set of non-overlapping line segments.
Critic	*Weighting*	The importance that Launchpad should place on the line distance critic vs. the component distance critic.
Coin Decoration	*# Coins in Group*	The number of coins that should be placed along a platform.
	Platform Length	Threshold platform length for coin placement. Any platform greater than this number will have coins placed along it.
	Coin over Gap Probability	The probability that a coin should be placed over a gap.

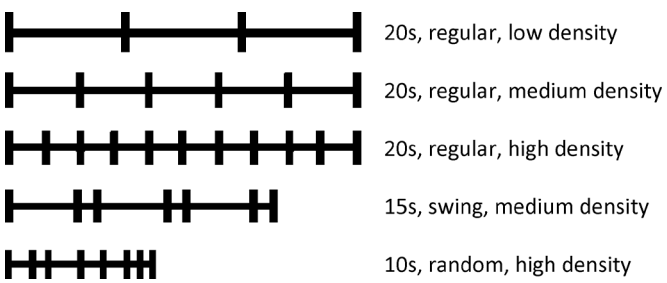

Fig. 7. Example rhythms. These timelines show the effects of varying the length, type, and density of a rhythm. Lines indicate the length of the rhythm, and hatch marks indicate the times at which an action will begin.

properties are shown in Fig. 7. Each beat of the rhythm marks a moment when the player takes an action by pressing a button.

Players can take two types of action: "move" and "jump." These actions also have an ending time, corresponding to when the player lets go of the button. This process produces rhythms like the one in the following, where the numbers after actions correspond to the begin and end times of the action. A graphical representation of this rhythm is shown in Fig. 8

$$\begin{array}{lcc} \text{move} & 0 & 5 \\ \text{jump} & 2 & 2.25 \\ \text{jump} & 4 & 4.25 \\ \text{move} & 6 & 10 \\ \text{jump} & 6 & 6.5 \\ \text{jump} & 8 & 8.5. \end{array}$$

In this example, the player starts moving at 0 s, and continues moving until 5 s. While moving, the player jumps once at the 2-s mark and again at the 4-s mark, with each jump lasting 0.25 s. The player then pauses moving at 5 s and begins moving again at 6 s until the end of the rhythm. While moving here, the player jumps once at the 6-s mark and again at the 8-s mark, this time holding down the "jump" button for longer than previously.

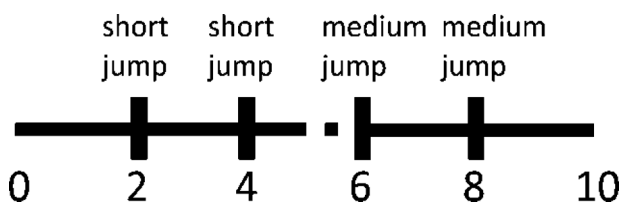

Fig. 8. Rhythm with actions. This timeline corresponds to the example rhythm given in Section IV-A. The solid line denotes time that the avatar is moving, the dashed line is time that the avatar is not moving.

The length associated with the "jump" verb corresponds to the amount of time the player will hold down the "jump" button. Since different hold times influence the height of the avatar's jump, it is important to keep these jump types distinct. For example, the player may hold the button down for only 0.25 s, resulting in a very short hop, or may hold the button down up to 0.75 s for a much longer jump.

B. Physics Constraints

The physics model maintains information about the avatar's capabilities and the different types of jumps available. The model includes the avatar's size, maximum movement speed, initial jumping velocity, and the height the avatar can jump given a short, medium, or long jump button press. For jumps, the model includes the in-air time for each jump type, the

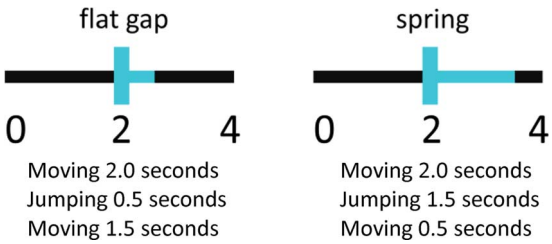

Fig. 9. Creating movement states. Two different types of jump can contribute to different movement and jump state lengths. The blue area is the amount of time consumed by the jump being in the air.

relative height difference for the two platforms on either side of a jump based on this in-air time, and the velocity imparted to the avatar by a spring. Available slopes for platforms are also recorded. This model is ballistics based, extended to allow variable jump heights due to different amounts of time the jump button is held. This style is common for Mario-style platformers. More advanced player physics, such as double jumping or wall jumping, are not supported.

All this information is used to process rhythms created by the rhythm generator into input suitable for the geometry grammar. In doing so, this guarantees that all levels are fully playable. Received verbs are first converted into a list of movement states and a queue of jump commands. This is done by examining the lengths of each action. Any time that the player is not "moving," they are considered to be "waiting." Jump lengths are categorized as either short, medium, or long based on the maximum amount of time that the jump button could be held down. For example, the example rhythm given above forms the following set of states and jump queue:

States: [5, moving], [1, waiting], [4, moving];

Jumps: [2, short], [4, short], [6, medium], [8, long].

However, the jump length merely tells how long the player holds the "jump" button; the actual time in the air varies based on which jump type is chosen. A jump across a flat gap takes considerably less airtime than a jump onto a spring. Therefore, jumps consume some amount of the movement state list at the time the jump occurs. For example, assuming there are only two jump options (flat gap and spring), a jump across a flat gap takes 0.5 s, and a jump onto a spring takes 1.5 s, processing the first jump in our example could result in one of the two configurations shown in Fig. 9. However, if the second jump were to occur at 3 s rather than 4 s, the jump onto a spring would no longer be a valid jump, as it would not end until after the next jump should begin. In this case, the spring would be disallowed from the set of geometry that could be used for that jump.

C. Geometry Realization

The movement states that are not fully consumed by jumping and the queued jumps form the nonterminals in the geometry generation grammar (Fig. 10). The "waiting" state is meaningless on its own, as there must be something to wait for. Generating geometry for a wait state therefore involves looking ahead in the state list. The geometry that can be chosen is confined by the physics constraints mentioned above, and is also influenced by the desired frequency of components as specified by the designer (Table I).

Moving → Sloped | flat_platform
Sloped → Steep | Gradual
Steep → steep_slope_up | steep_slope_down
Gradual → gradual_slope_up | gradual_slope_down

Jumping → flat_gap
| (gap | no_gap) (jump_up | Down | spring | fall)
| enemy_kill
| enemy_avoid
Down → jump_down_short | jump_down_medium
| jump_down_long

Waiting-Moving → stomper

Waiting-Moving-Waiting → moving_platform_horiz
| (moving_platform_vert_up | moving_platform_vert_down)

Fig. 10. Geometry generation grammar. Player states derived from the generated rhythms are the nonterminals in this grammar.

move	0.00	8.00
jump	2.00	2.25
jump	4.00	4.24
jump	6.00	6.25
move	10.00	12.00
move	14.00	20.00
jump	16.00	16.25
jump	18.00	18.25

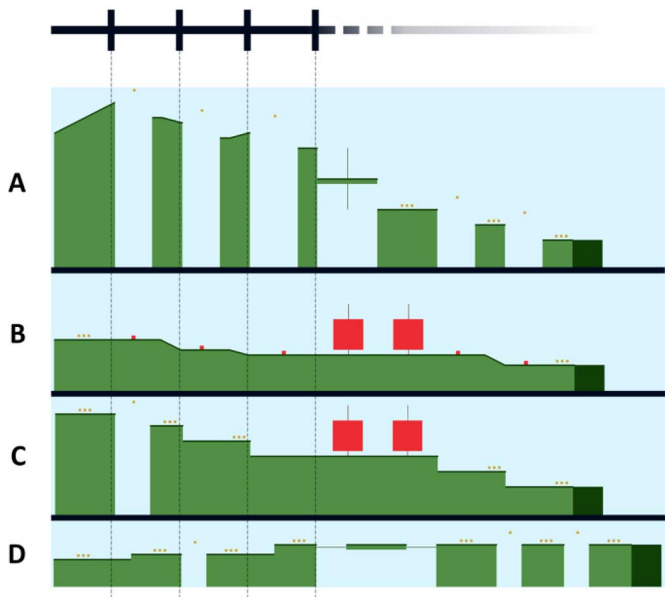

Fig. 11. Geometry interpretations of a rhythm. This figure shows four different geometric interpretations of the provided rhythm. Small red boxes denote enemies to kill, large red boxes are stompers that follow the associated line, and platforms on green lines are moving platforms that follow that path. The large, dark green platform at the end of the rhythm group is the joiner from this rhythm group to the next.

Fig. 11 shows an example of a rhythm and four different geometries that can be generated from it. Fig. 11(a) and (d) shows how moving platforms consume a wait–move–wait; other interpretations of two wait–moves in a row are shown in Fig. 11(b) and (c). The dotted vertical lines show how the first three jumps correspond to geometry; note that waits introduce variation to the physical length of the rhythm group and jumps

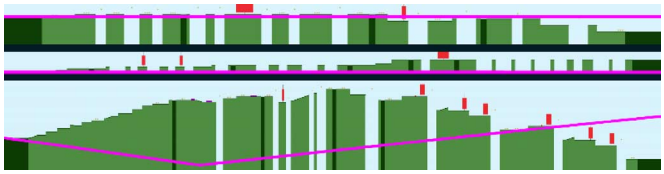

Fig. 12. Line critic scores. Examples of levels that are different distances from a specified control line (shown in pink). The top level has a distance measure of 1.75, the middle a distance measure of 5.08, and the bottom a distance measure of 23.06.

that occur after waits no longer "line up" with the sample rhythm. This figure also shows many different geometric interpretations for each jump action.

D. Critics

Complete levels are generated by piecing together rhythm groups that are connected via a joining platform. This gives the player an opportunity to rest before beginning the next challenge, an important aspect of level pacing [41]. Rhythm groups can optionally be repeated before this rest area, according to a probability that is set by the human designer. This can provide additional challenge [6] and more visual consistency. Each level is the length of the control line specified by the user, as described in the following.

Design grammars such as Launchpad's are good at capturing local constraints such as playability at each action point. However, design grammars also commonly lead to overgeneration: even with constraints on rhythms and geometry generation, the variety of levels created by Launchpad is extremely large. We use critics to narrow this space of levels according to more global heuristics: a control line (line distance critic) that the level should fit to, and the desired frequency of components (component frequency critic) appearing in the level. This allows a designer to focus Launchpad's output to a specific kind of level. A level designer can adjust the importance of each critic so as to exert some control over the kinds of levels that are produced.

1) Line Distance Critic: One input a human designer provides to the level generator is a path that the level should follow, providing control over where the level should start and end, and the general direction it should follow in between. The path is specified as a piecewise set of line segments. This critic serves to rein in the large space of levels that can be generated and allow the human designer to assert additional control; existing, human-designed levels in *Super Mario World* and *Sonic the Hedgehog* tend to follow regular paths, rather than meander aimlessly through space. The level that best fits this path minimizes the average distance between all platform endpoints and the path, as shown in Fig. 12.

2) Component Frequency Critic: The component parameters described in Table I are used in geometry generation as a weight for how often each component should be chosen; however, these weights only guarantee that over a large number of rhythm groups the frequency of each component will asymptotically approach the specified probability. Each rhythm group is created by pulling a relatively small number of geometric components from a pool of potential components, and then levels are created by again choosing a small number of rhythm groups from a large number of randomly generated rhythm groups. Therefore, there is no guarantee that the observed frequency of components will match the expected frequency, just as in a series of ten coin flips there is no guarantee that five will come up heads and five will come up tails even though one would expect 50% odds on each. The component frequency critic is designed to ensure that the number of each type of component in the level best matches the probability distribution formed by the style parameters. This critic works by applying a chi-square goodness-of-fit test [42] to each potential level, and choosing the level with the smallest test statistic. This represents the level that has the closest component distribution to the desired style.

3) Combining Critics: There are many cases in which these two critics will contradict each other as to which level is best. For example, a level that is heavily weighted towards having springs will not fit a line that slopes only downward. To resolve this contradiction, the level with the lowest weighted sum of the two critics is selected, where the weight on each critic describes its importance.

E. Global Passes

Although most of the level can be created with local generation techniques, it is important to be able to reason over these levels. Launchpad has two "global pass" algorithms: tying platforms to a common ground plane, and decorating levels with coins. These algorithms reason over both the player states and the geometry associated with them. Assigning platforms a common ground point, determined by the platform with the lowest y value, provides some visual consistency for levels and removes the possibility of the player unintentionally falling off platforms that are spaced far apart from each other.

Collectible items are treated as decoration over a level, in a manner stylistically consistent with *Super Mario World*. Two rules determine collectible item placement:

1) place a group of coins along a long platform of predetermined length that has no other action other than move associated with it;
2) reward the risk for jumping over a gap, and provide guidance for the ideal height of a jump, by placing a single coin at the peak of jumps that go over gaps.

The probabilities for these coins being placed, and the number of coins that should be placed, are specified by input parameters.

It would also be easy to specify new, powerful rules for coin placement, such as along the path of a spring or fall to guide the player in the right direction, due to Launchpad's ability to reason over rhythm and geometry independently. However, level generation for games that treat collectible items as a primary goal, such as *Donkey Kong Country*, would perhaps be better accomplished by placing coins during geometry generation, rather than as decoration after the fact.

V. Evaluation

Examples of the kinds of levels that Launchpad can create are shown in Fig. 13. However, these examples do not adequately show the range of levels that Launchpad can create, nor the full impact of changing generation parameters on this space. We call this quality of a generator its "expressive range,"

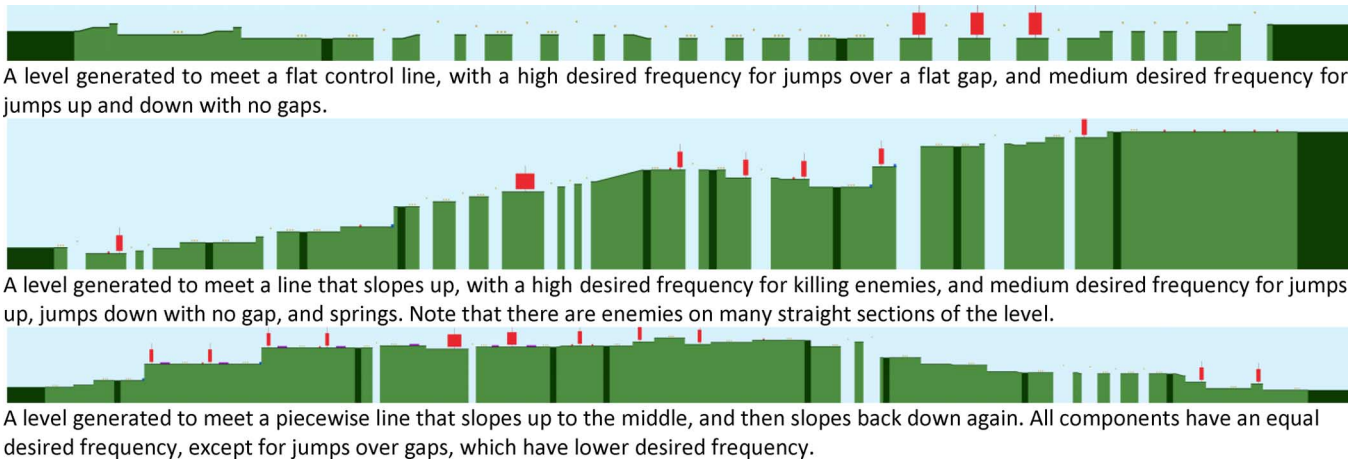

Fig. 13. Generated levels. A selection of levels generated by Launchpad using different generation parameters. Each level shown is the best fit to these parameters out of a sample size of 1000 candidate levels.

and have created a tool to visualize this range. This section describes Launchpad's expressive range by addressing the following questions.

1) How does the design of the generation algorithm itself affect the kinds of levels that can be produced?
2) What are appropriate ways to measure produced levels?
3) How does altering input parameters affect the qualities of generated levels?
4) How can we compare levels produced by the system?

A. Algorithm Implications

Certain kinds of platformer levels are excluded from Launchpad's expressive range due to our algorithm for level generation. In particular, we create levels that are dexterity based rather than exploration based. The challenge derived from these levels is more about perfectly timing movement through a series of obstacles, rather than seeking out hidden areas. Furthermore, Launchpad does not support the player choosing a path to take through the level, which is common in games like *Sonic the Hedgehog*, and do not support the player turning around. Because of this, Launchpad's levels tend to favor a "speed run" play style. This style influences our choice of comparison metrics for levels, discussed in the following.

B. Level Metrics

In order to describe the expressive range of a level generator, we must first be able to compare the levels that it produces. It is important that the metrics used for comparing levels measure emergent properties of levels, rather than simply reusing the same parameters used to guide the generator. In this way, we can see how input parameters affect the resulting levels. Based on the style of platformer that Launchpad creates levels for, we define two different metrics for generated levels: linearity and leniency. These metrics describe global qualities of levels, focusing on aesthetics (linearity) and gameplay (leniency).[1]

[1]Please note that these metrics differ from those published in our previous work. It is now the case that the higher the score, the better the level fits the associated metric. Also, the leniency metric has been normalized to $[0, 1]$.

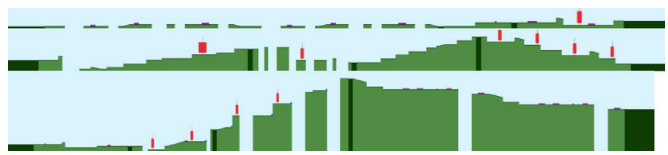

Fig. 14. Linearity scores. These three levels show decreasing linearity scores. The upper level has a score of 0.93, the middle a score of 0.60, and the bottom a score of 0.00.

Fig. 15. Linearity versus line distance. This level has a high linearity metric score, as it follows an upwardly sloping line. However, it has a low score for the line critic as it diverges significantly from the user-specified control line.

1) Linearity: Linearity measures the "profile" of produced levels; this is a more aesthetic quality that the player will experience while rushing through the level. We measure linearity by fitting a single line to the level and determining how well the geometry fits that line. The goal here is not to determine exactly what the line is, but rather to understand Launchpad's ability to produce levels that range between highly linear and highly nonlinear. Examples of levels that fall at the extremes of this scale are shown in Fig. 14. The linearity of a level is measured by performing linear regression, taking the center points of each platform as a data point. We then score each level by taking the sum of the absolute values of the distance from each platform midpoint to its expected value on the line, and divide by the total number of points. Results are normalized to $[0, 1]$, where 1 is highly linear and 0 is highly nonlinear (i.e., higher scores are a better fit). In our experiments, levels rarely had a linearity score lower than 0.3.

It is important to note that linearity and the line distance critic are two different things, and that it is possible for a level to be judged "highly linear" but have a poor line critic distance score (Fig. 15). The line distance critic is a measure for how well a level fits a control line specified by a designer before generation occurs, whereas linearity measures the overall linearity of

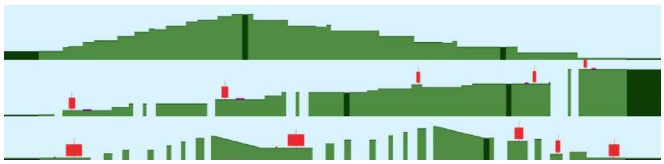

Fig. 16. Leniency scores. These three levels show decreasing leniency scores. The upper level has a score of 1, the middle a score of 0.5, and the bottom a score of 0.01.

a level that is output from the generator. Linearity is an aesthetic measure; line distance is a design control and heuristic.

2) Leniency: Leniency describes how forgiving the level is likely to be to a player. We hesitate to quantify the difficulty of generated levels, as this is a subjective measure and dependent on the specific ordering and combinations of components. However, it seems reasonable to describe levels that provide fewer ways for the player to come to harm as being more *lenient* than other levels. To measure this, we assign scores to each type of geometry that can be associated with a beat

-1.0 gaps enemies falls
-0.5 springs, stompers
$+0.5$ moving platforms
$+1.0$ jumps with no gap associated.

These scores are based on an evaluation of how lenient components are towards a player, with higher scores indicating more lenience. The overall leniency score is the sum of each score divided by the total number of components, then normalized. Example levels with different lenience scores are shown in Fig. 16.

C. Rhythm Group Distance Metric

In addition to objectively ranking entire levels using the linearity and leniency metrics, we also define a method for directly comparing two rhythm groups according to geometric similarity. We use these distance metrics in clustering rhythm groups, as described in Section V-D, so we can classify similarities and differences in the building blocks for our levels.

When comparing rhythm groups, the level of similarity or difference between them is computed by counting the edit operations that would be required to convert one rhythm group into another. This allows us to create a distance metric that is largely independent of the length of the rhythm groups, and can identify similarity between groups on a continuous scale. The distance metric used to compare rhythm groups is the Levenshtein edit distance [43] applied to vectors that code the type of geometry placed for each beat. This distance is normalized [44] to reduce the impact of the length of each rhythm on the distance between them. Table II provides a listing of the different encodings for different rhythm group geometry. Our distance metric uses an insertion and deletion cost of 2 and customized substitution costs. A substitution begins with a base cost of 0, and is modified in the following ways:

$+1$ for adding or removing a gap;
$+1$ for a moving platform;
$+1$ per step of changed vertical movement;
$+1$ for the appearance of an enemy or stomper;

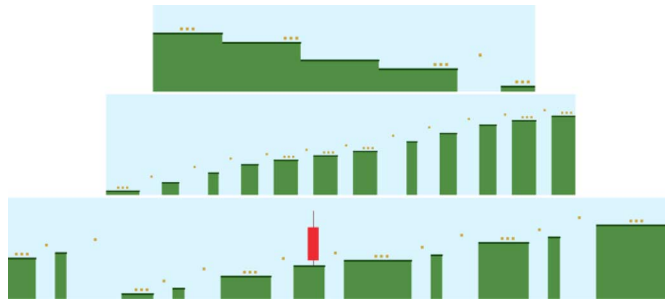

Fig. 17. Rhythm group distances. Distances between rhythm groups are normalized to $[0, 1]$. The distances between these three rhythm groups are as follows: $d(A, B) = 0.91$; $d(B, C) = 0.20$; $d(A, C) = 0.85$.

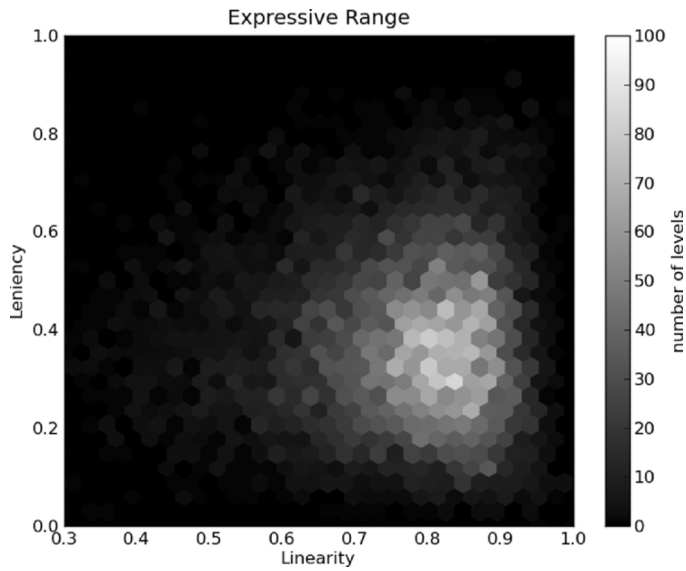

Fig. 18. Launchpad's expressive range. All geometry types are weighted equally. Linearity is measured on the x-axis, from 0.3 to 1.0. Leniency is measured on the y-axis, from 0 to 1.0. The color of each hexagon corresponds to the number of levels that have the associated linearity and leniency scores. The lighter the color, the more levels there are in that bin.

$+0.5$ for converting an enemy to stomper or *vice versa*.

These modifications are based on a similar motivation for the linearity and leniency metrics: we would like to see rhythm groups differ both in terms of visual aesthetics and gameplay differences. But by examining these differences at the level of individual geometry components, rather than as a global measure of a level, we can extract emergent patterns (e.g., staircases) in the rhythm groups that are produced. Example rhythm groups and the distances between them are shown in Fig. 17. Notice that the distance between rhythm groups B and C is fairly low, even though they have different lengths and some different geometry. The groups are more similar in terms of a pattern than they are different.

D. Expressive Range

These metrics allow us to compare produced levels and describe Launchpad's expressive range in two different ways: by categorizing entire levels into bins based on their linearity and leniency scores, and by clustering rhythm groups to see emergent patterns. Both approaches uncover the variety of levels that can be created with Launchpad, and the influence of parameters on these levels, but at different granularities.

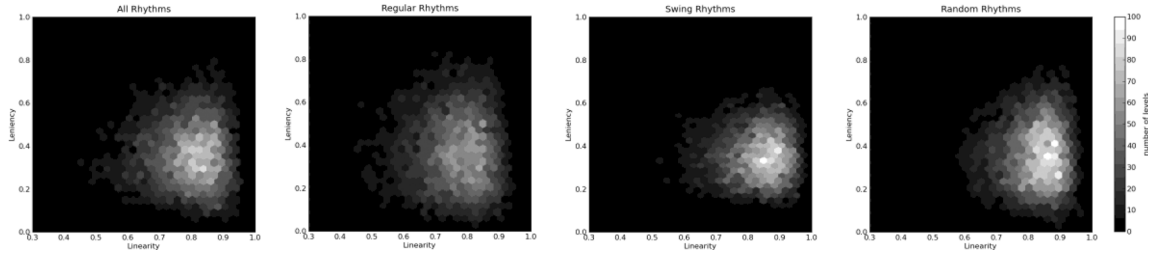

Fig. 19. Varying rhythm type. (A) All rhythms. (B) Only regular-type rhythms. (C) Only swing-type rhythms. (D) Only random-type rhythms.

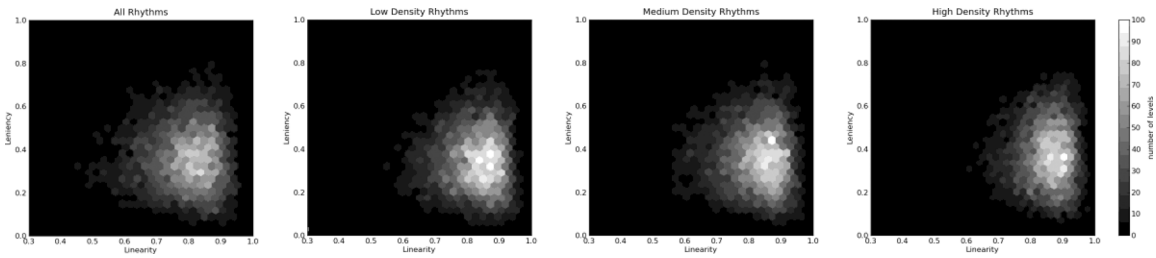

Fig. 20. Varying rhythm density. (A) All rhythms. (B) Only low-density rhythms. (C) Only medium-density rhythms. (D) Only high-density rhythms.

TABLE II
ABBREVIATIONS FOR LEVEL GEOMETRY

Code	Level Geometry
GA	A gap to a higher platform
MA	An ascending moving platform
A	A jump to a higher platform with no gap
GS	A gap to a platform of the same height
MS	A horizontally moving platform
GD	A gap to a lower platform
MD	A descending moving platform
D	A jump to a lower platform with no gap
E	An enemy
ST	A stomper

1) Level Analysis: We can describe the expressive range of Launchpad by generating a large number of levels and ranking them by their linearity and leniency scores. Fig. 18 shows the expressive range of the generator when all components are equally weighted and all rhythms are being used. Each hexagon is colored to indicate the number of generated levels that have the corresponding linearity and leniency scores. All graphs used in this paper are based on 10 000 generated levels, unless stated otherwise.

Here, the generative space is clearly biased towards more linear levels, and slightly biased towards less lenient levels (i.e., the lower right corner of the graph). The leniency bias is likely due to there being a greater number of nonlenient components available for selection. The linearity bias is a more interesting result, as we believe it is due to what was originally intended to be a small implementation detail in the level generator. When a component is chosen for inclusion in a rhythm group, the probability of that component appearing again is slightly increased. This detail was added late in the development of Launchpad, to fix a problem we perceived in our early levels: they did not have any discernable patterns, as we tend to see in games like *Super Mario World* where there tends to be locally repeated geometry. However, this approach means components that incur a height difference (e.g., jumping up to a new platform) are likely to stack up to create linear segments of the level. The linearity bias is an unintended side effect of this design decision. This issue highlights an important reason to perform a detailed analysis on a generator's expressive range: in addition to being able to easily display the variety of content, we can learn more about how rules interact inside the generator.

As noted in Table I, there are a number of different parameters that can be varied to change the kinds of levels that Launchpad produces. A crucial aspect of analyzing expressive range is to understand how varying these parameters affects the levels that are generated. For now, let us look at all levels that are created, rather than those that pass critic tests.

Fig. 19 shows the results of varying the rhythm type. The regular rhythm type offers the most variation of all the rhythm types, with no sharp peak in the graph. This distribution is what was initially expected for all rhythms; however, swing and random rhythms are more constraining, contributing heavily to the bias towards linear and nonlenient levels. One hypothesis is that these constraints are due to the potentially shorter amounts of time given to jumps in swing and random rhythms. This leads to the physics system filtering out potential geometry for beats due to the requirement for the avatar to land before the next jump occurs. For swing rhythms, this means that there will be fewer falls, springs, and moving platforms, contributing to a higher leniency score.

Fig. 20 shows the results of varying rhythm density. Varying this parameter does not have a noticeable impact on leniency, but higher densities do lead to more linear levels. This is for the same reason that the generator is biased towards creating linear levels; higher density rhythms have more actions in them, and each action has a higher probability of the same component being chosen as before. Fig. 21 shows similar results when varying rhythm length.

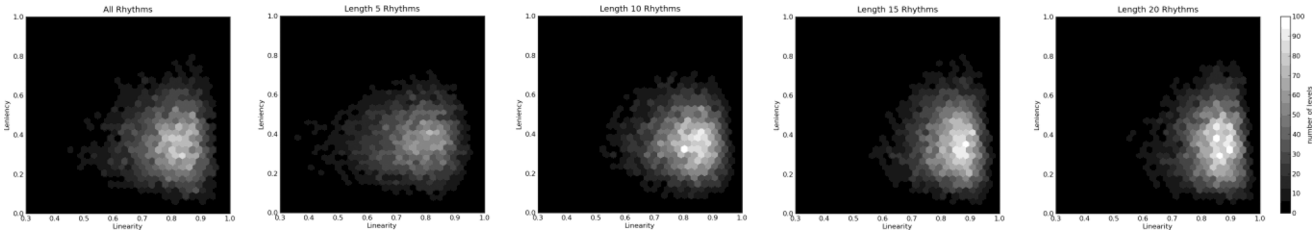

Fig. 21. Varying rhythm length. (A) All rhythms. (B) Only length 5 rhythms. (C) Only length 10 rhythms. (D) Only length 15 rhythms. (E) Only length 20 rhythms.

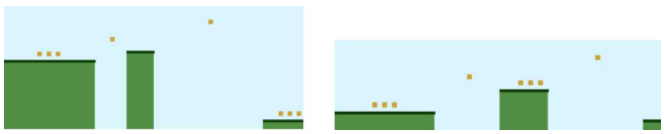

Fig. 22. Zero distance rhythm groups. Two rhythm groups whose edit distance is zero. Note that these groups are not identical, but do share the same pattern of jumping up over a gap, and then jumping down over a gap.

2) Rhythm Group Analysis: In addition to describing global qualities of levels, we can also cluster generated rhythm groups based on their similarity to each other. Visualizing these clusters provides a better sense of the variety of produced levels and shows patterns in produced geometry. We can also visualize how changing generation parameters influences the space of produced rhythm groups.

Rhythm groups are clustered using an agglomerative hierarchical clustering method [45]. The first grouping combines all rhythm groups that are interpreted as identical to each other. Note that these groups are not necessarily completely identical in terms of geometry placement; rather, they are identical according to their edit distance. An example of two rhythm groups that have zero distance between them is shown in Fig. 22. These bins of similar rhythm groups form the leaf nodes of our hierarchy, and are the initial clusters for the clustering algorithm. On each iteration of the algorithm, two clusters are selected to be grouped together. The heuristic used for this is the clusters whose most dissimilar members have the closest distance. Iteration continues until all initial clusters have been included in the hierarchy.

This cluster hierarchy can be visualized with a dendrogram; however, a static tree is difficult to navigate, and does not clearly show the size of each cluster or the contents of it. A better approach is to let a level designer navigate this tree using a treemap representation [46], where each box has a size corresponding to the number of rhythm groups contained in the cluster. Each cluster has a color assigned to it, with its children having a color of the same hue but different brightness. Initial coloring is determined by a distance parameter set by the designer: this distance parameter describes how far apart each cluster should be from one another.

Clustering is performed on 4000 rhythm groups: 1000 for each length of rhythm (Section IV-A). A side bar in the visualization tool shows the rhythm properties of each rhythm group in a selected cluster, allowing the user to see how rhythm parameters influence generated geometry. A region underneath the treemap shows thumbnails of all the rhythm groups contained

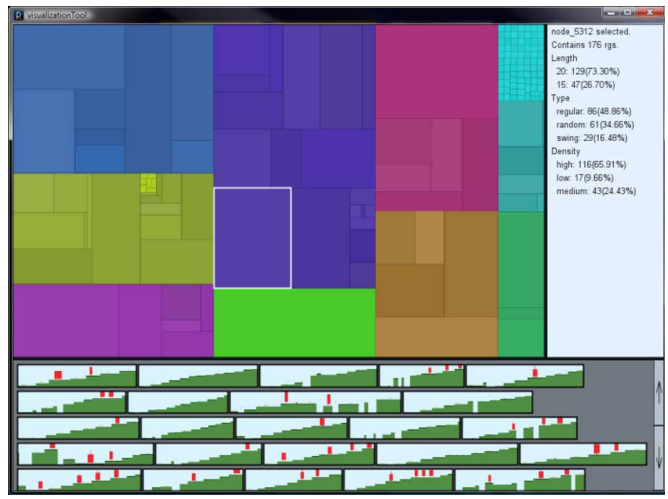

Fig. 23. Visualizing rhythm group clusters. The rhythm group cluster treemap at various depths. The user can navigate the tree, expanding and collapsing nodes. At each stage, the area below the treemap shows the rhythm groups contained in the selected cluster.

in the cluster, allowing easy visualization of variety within a cluster. Fig. 23 shows a screenshot of the cluster visualization tool.

Exploring each cluster gives a sense of the large range and variety of content that Launchpad can create. Like the expressive range graphs, this visualization has also helped to uncover some biases in Launchpad's generation algorithm. The largest size clusters correspond to rhythm groups that contain a lot of stompers. There are also a large number of rhythm groups that contain geometry that forms an upward staircase pattern. The discovery of an upward staircase pattern offers additional evidence for the theory mentioned earlier that shorter jump times lead to the physics system filtering out jumps that might cause the player to move to a lower y position. The prevalence of stompers is likely due to how Launchpad handles the player waiting. Recall that stompers and moving platforms can be created for wait–move and wait–move–wait beat patterns, respectively. However, the wait–move–wait pattern also contains a wait–move, meaning the stomper can be chosen in that situation as well. This biases the generator towards creating a larger number of stompers than expected.

VI. Discussion

This paper has described Launchpad, a level generator for 2-D platformers built on a rhythm-based model of player behavior, derived from an analysis of existing platformer games. It

also presents a visualization tool for understanding Launchpad's expressive range. Both the Launchpad demo and the visualization tool are available online.[2] The demo includes 15 sets of pregenerated levels, where each set has different desired component frequencies, control lines, and player movement speed. It also has a standalone "generate" feature, which will create a single level with a random player speed.

While rhythm and pacing are important aspects of platformers, they are not the sole determining factors of an enjoyable level. With this in mind, we initially considered a difficulty-based approach for level generation, rather than our current rhythm-based approach. This method would have involved assigning difficulty measures to different, large-scale "idioms" for platformer levels, such as jumping onto a platform with a moving enemy, or jumping across a series of variable-width gaps. These idioms would then be fit together in much the same manner as existing level generation techniques, but with heuristics for controlling the difficulty of each chunk. However, we were concerned that this approach would not provide sufficiently varied levels, nor be as extensible. The rhythm-based approach provides more flexibility by working with the most basic components of levels, rather than their myriad combinations, and recognizes the structural importance of rhythm in platformers. We feel that it is both simpler and more beneficial to build levels using a well-defined structure and later analyze them for difficulty, than *vice versa*. Polymorph [31] is a project that uses this approach to level generation, assigning difficulty to individual rhythm groups based on statistics gathered from players.

Another critical design decision in creating Launchpad was using a generate-and-test process over candidate levels rather than carefully piecing together rhythm groups using a more sophisticated search process. The rationale for this decision is to make it easy for a designer to use Launchpad to rapidly view many different possibilities for a level that meet designer-specified criteria, and because these criteria are global in nature. It is convenient to have a large pool of candidate levels ranked by their fits to these design heuristics and allow the designer to browse that pool. However, if the design criteria became more sophisticated, for example by allowing a designer to specify the kinds of geometry in particular regions of a level, then a search-based stitching together of rhythm groups may become more appropriate, as the number of candidate levels required to provide a good match to increasingly complicated design criteria would be prohibitive.

There are a number of different directions that future work on Launchpad could take. Perhaps the simplest extension would be the addition of new components to the system by creating new verbs (e.g., "shoot") and geometry associated with those verbs. However, it is important to note that this extension may also require modifications to the rhythm generator, to ensure that these new verbs and geometry could be chosen. Our expressive range evaluation has shown that Launchpad currently favors placing patterns of components that take smaller amounts of time for the player to cross, due to rhythm constraints. Therefore, new components that require longer amounts of time to cross (e.g., a loop-de-loop from *Sonic the Hedgehog*) may be chosen less frequently than desired without also performing tuning on the rhythm generator. Since Launchpad does not contain any search processes, the addition of new components would not slow down level generation.

Other components, such as triggers that affect the rules or physics of the game, may be more challenging to add. Launchpad can guarantee level playability by creating locally playable components and connecting them with safe walking areas. This no longer works when generating levels for a game like *Shift*, where the physics properties of the game change at runtime. A grammar-based approach may not be appropriate for generating levels with such characteristics.

Finally, we would like to explore how Launchpad's generation technique can extend to other genres. Dormans's work [23] in creating missions and levels for *Zelda*-style adventure games offers encouraging results in grammar-based level generation for a different genre. However, we believe that more important than a specific generation technique is the focus on understanding and reasoning about expected player actions in a level. In Launchpad, this is represented as player actions occurring at specific beats; in another genre, this may be a quest that the player is following, or skills that the player should be learning.

This paper has presented Launchpad, a level generator for 2-D platform games that works from a formal understanding of platformer level design. Launchpad provides designers with control over the kinds of levels it produces through a set of parameters that have a clear mapping to the generated levels. Furthermore, this paper describes a method for analyzing and visualizing the expressivity of a level generator, and showed that Launchpad has a wide expressive range within its domain. Playable examples of levels generated with Launchpad and the visualization tool are available online.

[2] Launchpad: http://users.soe.ucsc.edu/~gsmith/launchpad/platformer/. Visualization tool: http://users.soe.ucsc.edu/~gsmith/launchpad/viztool/

References

[1] E. Byrne, *Game Level Design (Game Development Series)*. Boston, MA: Charles River Media, 2004, pp. 1–12.

[2] Nintendo EAD, Super Mario World, 1990.

[3] Sonic Team, SEGA, Sonic the Hedgehog, 1991.

[4] G. Smith, M. Cha, and J. Whitehead, "A framework for analysis of 2D platformer levels," in *Proc. ACM SIGGRAPH Sandbox Symp.*, Los Angeles, CA, 2008, pp. 75–80.

[5] K. Compton and M. Mateas, "Procedural level design for platform games," in *Proc. 2nd Artif. Intell. Interactive Digit. Entertain. Conf.*, Palo Alto, CA, 2006, pp. 109–111.

[6] V. Nicollet, "Difficulty in dexterity-based platform games," *GameDev.net*, Mar. 2004 [Online]. Available: http://www.gamedev.net/reference/design/features/platformdiff

[7] Side Effects Software, Houdini 11 (PC Software), 2010.

[8] Procedural Inc., CityEngine (PC Software), 2010.

[9] Interactive Data Visualization Inc., SpeedTree (PC Software), 2010.

[10] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*. Cambridge, MA: MIT Press, 2004, pp. 312–327.

[11] P. Co, *Level Design for Games: Creating Compelling Game Experiences*. Berkeley, CA: New Riders Games, 2006.

[12] E. Adams, *Fundamentals of Game Design*. Berkeley, CA: New Riders Press, 2009.

[13] J. H. Feil and M. Scattergood, *Beginning Game Level Design*. Boston, MA: Thompson Course Technology, 2005.

[14] M. Nelson, "Breaking down breakout: System and level design for breakout-style games," *Gamasutra*, Aug. 2007 [Online]. Available: http://www.gamasutra.com/view/feature/1630/breaking_down_breakout_system_and_.php

[15] K. Hullett and J. Whitehead, "Design patterns in FPS levels," in *Proc. Int. Conf. Found. Digit. Games*, Monterey, CA, 2010, pp. 78–85.

[16] D. Milam and M. S. El Nasr, "Analysis of level design 'Push & Pull' within 21 games," in *Proc. Int. Conf. Found. Digit. Games*, Monterey, CA, 2010, pp. 139–146.

[17] D. Boutros, "A detailed cross-examination of yesterday and today's best-selling platform games," *Gamasutra*, Aug. 2006 [Online]. Available: http://www.gamasutra.com/view/feature/1851/a_detailed_cross-sexamination_of_.php
[18] J. Dormans, "The art of jumping," Nov. 2005 [Online]. Available: http://www.jorisdormans.nl/article.php?ref=artofjumping
[19] M. Toy, G. Wichman, K. Arnold, and J. Lane, *Rogue*, 1980.
[20] Blizzard North, Blizzard Entertainment, Diablo, 1997.
[21] Rogue Basin, Articles on Implementation Techniques, [Online]. Available: http://roguebasin.roguelikedevelopment.org/index.php?title=Articles#Implementation
[22] D. Yu, Spelunky, 2009 [Online]. Available: http://www.spelunky-world.com/
[23] J. Dormans, "Adventures in level design: Generating missions and spaces for action adventure games," in *Proc. Workshop Procedural Content Generat. Games*, Monterey, CA, 2010.
[24] G. Stiny, "Introduction to shape and shape grammars," *Environment Planning B*, vol. 7, no. 3, pp. 343–351, 1980.
[25] C. Ashmore and M. Nitsche, "The quest in a generated world," in *Proc. Digit. Games Res. Assoc. Conf., Situated Play*, Tokyo, Japan, 2007, pp. 503–509.
[26] P. Mawhorter and M. Mateas, "Procedural level generation using occupancy-regulated extension," in *Proc. IEEE Conf. Comput. Intell. Games*, Copenhagen, Denmark, 2010, pp. 351–358.
[27] K. Hullett and M. Mateas, "Scenario generation for emergency rescue training games," in *Proc. Int. Conf. Found. Digital Games*, Orlando, FL, 2009, pp. 99–106.
[28] J. Togelius, R. De Nardi, and S. M. Lucas, "Towards automatic personalised content creation for racing games," in *Proc. IEEE Symp. Comput. Intell. Games*, Honolulu, HI, 2007, pp. 252–259.
[29] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Automatic content generation in the galactic arms race video game," *IEEE Trans. Comput. Intell. AI Games*, vol. 1, no. 4, pp. 245–263, Dec. 2009.
[30] N. Shaker, G. N. Yannakakis, and J. Togelius, "Towards automatic personalized content generation for platform games," in *Proc. 6th Artif. Intell. Interactive Digit. Entertain. Conf.*, Palo Alto, CA, 2010, pp. 63–68.
[31] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: A model for dynamic level generation," in *Proc. 6th Artif. Intell. Interactive Digit. Entertain. Conf.*, Palo Alto, CA, 2010, pp. 138–143.
[32] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 669–677, 2003.
[33] A. Smith, M. Romero, Z. Pousman, and M. Mateas, "Tableau machine: A creative alien presence," in *Proc. AAAI Spring Symp. Creative Intell. Syst.*, Palo Alto, CA, 2008, pp. 82–89.
[34] Sonic Team, SEGA, Sonic the Hedgehog 2, 1992.
[35] Artoon, Nintendo, Yoshi's Island DS, 2006.
[36] Rareware, Nintendo, Donkey Kong Country 2: Diddy's Kong Quest, 1995.
[37] Nintendo EAD, New Super Mario Bros., 2006.
[38] H. Desurvire, M. Caplan, and J. A. Toth, "Using heuristics to evaluate the playability of games," in *Proc. CHI'04 Extended Abstracts on Human Factors Comput. Syst.*, Vienna, Austria, 2004, pp. 1509–1512.
[39] Armor Games, Shift, 2008 [Online]. Available: http://armorgames.com/play/751/shift
[40] C. Bleszinski, "The art and science of level design," 2000 [Online]. Available: http://www.cliffyb.com/art-sci-ld.html
[41] R. Kremers, *Level Design: Concept, Theory, and Practice*. Boca Raton, FL: Peters/CRC Press, 2009, pp. 263–267.
[42] G. W. Snedecor and W. G. Cochran, *Statistical Methods*. Ames, IA: Iowa State Univ. Press, 1989, pp. 76–79.
[43] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.
[44] L. Yujian and L. Bo, "A normalized Levenshtein distance metric," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1091–1095, Jun. 2007.
[45] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge Univ. Press, 2008, pp. 346–368.
[46] B. Fry, *Visualizing Data: Exploring and Explaining Data With the Processing Environment*. Sebastopol, CA: O'Reilly Media, 2008, pp. 182–219.

Gillian Smith (S'10) received the B.S. degree in computer science from the University of Virginia, Charlottesville, in 2006 and the M.S. degree in computer science from the University of California Santa Cruz, Santa Cruz, in 2009, where she is currently working towards the Ph.D. degree in computer science.

She teaches game design and programming in the University of California Santa Cruz COSMOS summer program for high school students. Her research interests include procedural content generation and mixed-initiative design tools.

Ms. Smith is a student member of the Association for Computing Machinery (ACM), the Association for the Advancement of Artificial Intelligence (AAAI), and the International Game Developers Association (IGDA).

Jim Whitehead (S'94–M'06–SM'08) received the Ph.D. degree in information and computer science from the University of California Irvine, Irvine, in 2000.

He is an Associate Professor at the Computer Science Department, University of California Santa Cruz, Santa Cruz. He was an active participant in the creation of the Computer Science: Computer Game Design major at the University of California Santa Cruz in 2006. His research interests include software evolution, software bug prediction, procedural content generation, and augmented design.

Prof. Whitehead is a member of the Association for Computing Machinery (ACM) and the International Game Developers Association (IGDA). He is the founder and chair of the Society for the Advancement of the Science of Digital Games (SASDG).

Michael Mateas received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 2002.

He is an Associate Professor at the Computer Science Department, University of California Santa Cruz, Santa Cruz, where he holds the MacArthur Endowed Chair. His research is in AI-based art and entertainment combines science, engineering, and design to push the frontiers of interactive entertainment. He founded and co-directs the Expressive Intelligence Studio at the University of California Santa Cruz, which has ongoing projects in autonomous characters, interactive storytelling, game design support systems, procedural content generation, automated game design, and learning AI from data-mining gameplay traces. With A. Stern, he released *Façade*, the world's first AI-based interactive drama. *Façade* has received significant attention, including top honors at the Slamdance independent game festival.

Mike Treanor received the B.S. degree in computer science and the MFA degree in digital arts and new media from the University of California Santa Cruz, Santa Cruz, in 2006 and 2008, respectively, where he is currently working towards the Ph.D. degree in computer science.

His research interests include experimental art games, social games, news/rhetoric games, and procedural content generation.

Jameka March is currently working towards the B.S. degree in computer science: computer game design at the University of California Santa Cruz, Santa Cruz, which she expects to receive in 2012.

Mee Cha received the B.S. degree in computer science: computer game design from the University of California Santa Cruz, Santa Cruz, in 2010.

She is currently with OnLive, Palo Alto, CA.