UNIVERSITY OF CALIFORNIA

SANTA CRUZ

EXPRESSIVE DESIGN TOOLS: PROCEDURAL CONTENT GENERATION FOR GAME DESIGNERS

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Gillian Margaret Smith

June 2012

The Dissertation of Gillian Margaret Smith
is approved:

_____
Professor Jim Whitehead, Chair

_____
Associate Professor Michael Mateas

_____
Associate Professor Noah Wardrip-Fruin

_____
Professor R. Michael Young

_____
Tyrus Miller
Vice Provost and Dean of Graduate Studies

TABLE OF CONTENTS

List of Figures ..... **ix**

List of Tables ..... **xvii**

Abstract ..... **xviii**

Acknowledgments ..... **xx**

Chapter 1: Introduction **1**

1 Procedural Content Generation 6

1.1 Game Design 7

1.2 Procedural Content Generation for Game Design 9

2 2D Platformer Games 13

3 Research Contributions 15

3.1 Design Understanding 16

3.2 Control 16

3.3 Intersecting the Design Process 16

3.4 Expressivity 19

4 Dissertation Overview 20

Chapter 2: Procedural Content Generation for Game Design **23**

1 The Design of PCG Systems 23

1.1 Controlling the Generator's Expressivity 24

1.2 Understanding Game Design and Players 34

2 Procedural Content Generation for 2D Platformer Games 36

3 AI in Tools for Design and Creativity 43

Chapter 3: Design Space Analysis 49

1 Existing Models of Level Design 50

2 Deconstructing Platformers 52

3 The Composition of Platformers 56

3.1 The Avatar 56

3.2 Level Components 57

4 The Rhythm-Based Structure of Platformer Levels 59

5 Case Study: Sonic the Hedgehog 63

6 Discussion and Future Work 66

Chapter 4: Launchpad: Designer-Influenced Level Generation 69

1 Introduction 69

2 Generating Levels 72

2.1 Rhythm Group Generation 73

3 Critics 79

3.1 Line Distance Critic 80

3.2 Component Frequency Critic 81

3.3 Combining Critics 82

4 Global Passes 82

5 Discussion and Future Work 83

5.1 Future Work 87

Chapter 5: Tanagra: Mixed-Initiative Level Generation 88

1 Related Systems 90

2 Level Representation 93

2.1 Beats 94

2.2 Geometry Pattern Library 95

3 Design Environment 100

3.1 Geometry Editing 101

3.2 Beat Editing 102

3.3 Playtesting 103

3.4 Generator Invocation 103

4 System Overview 105

4.1 Reactive Planning with ABL 107

4.2 Numerical Constraint Solving with Choco 110

5 Geometry Management and Generation 113

5.1 Hierarchical Geometry Patterns 113

5.2 Creating User Geometry from Tiles 119

5.3 Incorporating User-Created Geometry into Patterns 120

6 Beat Management 122

7 Constraint Solving and Search 124

7.1 Constraint Solving 125

7.2 Searching for a Solution 126

8 Use Scenario 127

9 Interface Changes 133

10 Reactive Grammars 136

11 Future Directions 139

Chapter 6: Endless Web: PCG-Based Game Design 143

1 Introduction 143

2 Endless Web 146

2.1 PCG Influence on Story 153

3 The Use of PCG in Game Design 153

3.1 Replayability 154

3.2 Adaptability 155

3.3 Player Control 156

3.4 Game Mechanics 158

3.5 *Endless Web* as a PCG-Based Game 158

4 The PCG-Based Game Design Process 160

4.1 The Influence of Domains 162

4.2 Entering the Design Loop 163

4.3 Design Challenges 164

5 Designing Endless Web 165

5.1 Early Prototypes 165

5.2 Modifications to Launchpad 170

6 Design Challenges 172

6.1 Using PCG as a Mechanic 172

6.2 Balancing Control for a Generative Space 173

6.3 Navigating a Generative Space 175

6.4 Teaching the Player 179

6.5 Art Influence and Challenges 180

6.6 Technical Design Decisions 181

7 General Lessons 183

8 Discussion 185

Chapter 7: Evaluating the Expressive Range of PCG Systems 187

1 Expressive Range 187

2 Measuring and Comparing Levels 189

2.1 Algorithm Implications 189

2.2 Level Metrics 190

2.3 Comparing Levels 193

3 Expressivity Analysis for Launchpad 195

3.1 Level Analysis 195

4 Expressivity Analysis for Tanagra 207

4.1 Shaping the Generative Space 209

5 Discussion 215

Chapter 8: Conclusions and Future Work 217

1 Summary of Contributions 217

2 Future Work 219

2.1 Engaging Computers as Creative Equals 219

2.2 Creating New Game Genres 223

2.3 Tools for New Domains 224

3 Conclusion 226

References 228

LIST OF FIGURES

Figure 1. **Screenshots from *Wizardry* and *Dragon Age: Origins*.** Advances in computer graphics technology mean that the player is exploring a more detailed and immersive world in *Dragon Age*, but the gameplay of the two games is very similar. 3

Figure 2. **Two examples of 2D platformer games.** Left: a portion of the Yoshi's Island 2 map from *Super Mario World*. Right: A portion of the Marble Zone I map from *Sonic the Hedgehog*. These two games have almost identical core mechanics, yet very different styles of level design. 14

Figure 3. **A level being designed with Tanagra.** 17

Figure 4. **Screenshots from *Endless Web*.** These screenshots showi examples of two different configurations of *Endless Web*'s generative space. 18

Figure 5. **Three Worlds from Super Mario Bros. showing different kinds of player behavior.** Top: World 1-2; Middle: World 3-4; Bottom: World: 6-3. World 1-2 involves the player mostly running and jumping; World 3-4 has the player take mostly *wait* actions; World 6-3 involves a combination of jumping and waiting. Maps are courtesy http://www.vgmaps.com 54

Figure 6. **A diagram of the rhythm-based level framework.** This conceptual model shows how the components in a level fit in to a more abstract struture. The thick-border boxes highlighted in blue are implemented in the systems described in this dissertation. 55

Figure 7. **Example uses of collectible items.** Left: 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 four 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. Right: Collectible bananas literally point the way to a hidden area in *Donkey Kong Country 2*. 59

Figure 8. **Trigger use in *Super Mario World.*** 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. 59

Figure 9. **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. 61

Figure 10. **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. 62

Figure 11. **Structural Analysis of Sonic the Hedgehog 2.** This is a segment of a level from *Sonic the Hedgehog 2*, split into rhythm groups marked by solid black lines and cells marked by background colors. 64

Figure 12. **Launchpad architecture diagram.** Blue boxes denote artifacts produced by the system; green ovals represent components of the generator, each of which is influenced by design parameters. 71

Figure 13. **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. 73

Figure 14. **Rhythm with actions.** This timeline corresponds to the example rhythm specified above. The solid line denotes time that the avatar is moving, the dashed line is time that the avatar is not moving. 74

Figure 15. **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. 76

Figure 16. **Geometry generation grammar.** Player states derived from the generated rhythms are the main non-terminals in this grammar, bolded here. Other, intermediate non-terminals (italicized) are used for expressing further variety in different types of geometry components. 77

Figure 17. **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. 78

Figure 18. **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. 81

Figure 19. **A screenshot from Launchpad's online demonstration.** This is one of 10 levels that fits the parameters and control line specified by a designer. 84

Figure 20. **Tanagra level components.** These are the tiles used by Tanagra. Each level component is made up of one or more of these tiles. 93

Figure 21. **A grammar-like representation of Tanagra geometry patterns**. Numbers in parentheses correspond to the number of beats required (on the left hand side) or the number of the beat to fill with a pattern (on the right hand side). 95

Figure 22. **Example instantiations of the four single-beat geometry patterns used in Tanagra.** (A) A gap between two platforms, (B) a stomper, (C) an enemy, and (D) a spring to a different platform. Patterns are described and produced using ABL behaviors. Note that there are many different configurations of each pattern; the precise placement of geometry is determined by the constraint solver (Choco). 96

Figure 23. **Example instantiations of the four multi-beat geometry patterns used in Tanagra.** (A) A gap followed by an enemy, (B) a staircase, (C) a valley,and (D) a mesa. 97

Figure 24. **Illustration of constraints placed by a Staircase pattern on gaps and platforms.** The three gaps are constrained to have the same width and height as each other, and each gap's width and height constraints the relative positions of their surrounding platforms within a beat. 99

Figure 25. **The Tanagra design environment.** The level canvas in the top right is where the level is drawn by both the human and computer. A beat timeline along the bottom allows the designer to control level pacing independently from geometry. Controls on the left side allow the player to generate a level, playtest the level, and draw in geometry. 100

Figure 26. **The Tanagra tool in playtesting mode.** The avatar is represented by the robot; a yellow bar inside the beat timeline shows where in the level the avatar is located..... 104

Figure 27. **Tanagra main architecture diagram.** Tanagra is made up of three main components: the GUI, an ABL agent, and the Choco constraint solver. The GUI and ABL communicate through working memory. ABL posts constraints to Choco and determines when the solver should be called; Choco responds with either a potential solution or a notification that no solution exists. A library of geometry patterns are specified using ABL behaviors..... 105

Figure 28. **Tanagra working memory knowledge representation hierarchy.** The hierarchy of working memory elements used to represent levels in Tanagra. Solid, black arrows denote inheritance; dashed blue arrows denote ownership. For example, *ConstraintWME*s have multiple owners, each of which must be a *ConstrainableWME*. 108

Figure 29. **Tanagra management and communication diagram.** This diagram shows the communication channels between working memory, ABL, and Choco, and dependencies between different managers within ABL. Green solid lines denote WME reading and writing, and orange dashed lines denote behavior and function calling. 109

Figure 30. **A diagram representing a potential partial level during level generation.** Beats A and B are consumed by the Valley Pattern, which in turn is a gap pattern, enemy pattern, and another gap pattern. The final beat in the level is assigned a spring pattern, which

places two platforms, a spring, and a gap. Blue, rounded boxes denote ABL behaviors. Green boxes are WMEs. Not represented here are the tiles that make up each level component. Table 2 provides examples of the constraints imposed between each level component. .. 117

Figure 31. **Correcting beat placement.** This figure illustrates the necessity for correcting beat placement based on geometry placement. The blue shaded boxes and dashed lines denote beat boundaries. If the designer draws geometry as depicted on the left, there can never be a solution to the level as the platforms do not line up at the beat boundary. However, this geometry placement should clearly be playable. The right side shows how Tanagra corrects beat boundaries to address this issue. 123

Figure 32. **The necessity for constraint relaxation.** This example shows how constraint relaxation is necessary when creating levels. On the left, a level with a user-specified platform on the right is being generated. A gap pattern has been chosen for the first beat, and Choco has instantiated this pattern as a jump down. In the middle example, the second beat also has a gap pattern, but using the existing constraints for the first beat, the level is unplayable. The right example shows how relaxing the positioning constraints for the first beat renders the level playable, without changing any of the geometry patterns. 124

Figure 33. **The necessity for geometry search.** A small example illustrating the need for geometry search. On the left, a partial level is being solved, where existing geometry in the first and third beats is pinned in place by the user. A geometry pattern must be selected for the middle beat. In the middle level, Tanagra selected the stomper pattern to fill the middle beat, which leads to an unsolvable level, as the exit platform and entry platforms for beats two and three can never line up. On the right, Tanagra has instead selected the spring pattern, which leads to a playable level segment. 124

Figure 34. **Tanagra use case: full level generation.** An initial generated level provided by Tanagra. 128

Figure 35. **Tanagra use case: user-drawn geometry.** Tanagra has automatically split the early beats in the level to accommodate the extra platforms. 129

Figure 36. **Tanagra use case: auto-filling geometry (example 1).** The designer has asked Tanagra to auto-fill the remainder of the level with generated geometry. User-drawn and pinned platforms are shown darker than generated platforms. 129

Figure 37. **Tanagra use case: auto-filling geometry (example 2).** A second potential level that fits the user-specified geometry. 130

Figure 38. **Tanagra use case: moving geometry.** The designer moves the end platforms higher in the level. 131

Figure 39. **Tanagra use case: rhythm changes (example 1)**. There are more beats in the middle and fewer at the beginning and end. 131

Figure 40. **Tanagra use case: rhythm changes (example 2).** Further rhythm changes are made by deleting beats at the center of the level. 132

Figure 41. **Tanagra use case: selective geometry generation.** Finally, the designer regenerates geometry for only the third and second to last beat. 132

Figure 42. **A screenshot showing an updated interface for Tanagra.** Geometry pattern preference toggles decorate the beat timeline, which is now shaded differently every other beat. Anchor points at beat boundaries can be dragged up and down to manipulate platform positions. The left side of the screen has been replaced with a simpler interface with three buttons: generate, reset, and play. 134

Figure 43. **Geometry preference toggles.** Toggles are in place for each geometry pattern: for enemies, stompers, gaps, springs, and falls. Yellow denotes a neutral preference, green denotes a preference, and red denotes undesirability. 135

Figure 44. **The six different kinds of tuning portals**. Left to right: *enemies/conflict, platform hazards/betrayal, springs/entering the unknown, gaps/failure, moving platforms/losing control*, and *stompers/stress*. 147

Figure 45. **Three different configurations of the Dream**. The leftmost screenshot shows two stompers and an enemy patrolling the long, unbroken platform. The middle shows a world with a lot of springs from all the tiers of difficulty. The rightmost has platform hazards and gaps. The background color of the world shifts colors to reflect the predominant challenges and difficulty tiers. 148

Figure 46. **The Web visualization.** A visualization of all the different goals available to the player that serves as a map of the generative space; thick lines denote tier boundaries, thin lines denote the spaces between tiers. The player has six primary goals to collect the dreamers (upper left) and six secondary goals to collect the powerups (lower left). The colored circles on each strand of the Web signifies the player's current location. When the player mouses-over an icon, all identical icons are highlighted. 149

Figure 47. **Themed art assets for *Endless Web.*** Top: The seven different art assets for non-hazardous platforms. The top asset is for the undisrupted Eidolon world, the remainder correspond to the nightmares (top to bottom): *Body Horror, Broken Hearts, Creepy Crawlies, Dolls and Roses, Faceless Crowds*, and *Time and Death*. Bottom: The seven different art assets for normal springs, in the same nightmare order as the platform assets. 152

Figure 48. **Run-time art asset selection.** As the player approaches a dreamer, the art style slowly coalesces to that of the nightmare (left: near the *Time and Death* nightmare). When further away from any one dream, the art is more jumbled (right: a level showing art for *Body Horror*, *Broken Hearts*, and *Creepy Crawlies*). 152

Figure 49. **The AI-based game design process.** The inputs to the process are domain knowledge relevant to both the game and the AI system, and the output is a playable

experience. AI Design and Game Design are iterated on and inform each other throughout the process, especialy with a view to resolving two key challenges in AI-Based Game Design: Transparency of the AI and handling emergent gameplay..... 161

Figure 50. **Early *Endless Web* technical prototype.** The technical prototype built to understand the impact of altering parameters on the player's experience. Sliders on the right control every parameter of the generator. The area on the left allows the designer to play levels as soon as they are generated. 166

Figure 51. **A screenshot from *Rathenn*, the early playable prototype for *Endless Web*.** The darker colored level geometry is the current playable level; the lighter geometry is a path that the player could have taken but chose not to. The black rectangle in the center of the screen is the player, multi-colored boxes are coins, the large red rectangle on the left is a stomper, and there are ladders along a choice-platform visible in the backgrounded level. 167

Figure 52. ***Endless Web* concept art.** A selection from the 54 original concept art sketches for the spikes challenge. 180

Figure 53. **Launchpad linearity example.** Examples of levels generated by Launchpad with the highest (top) and lowest (bottom) linearity scores. 190

Figure 54. **Linearity vs. Line Critic.** An example of a level generated by Launchpad with a high *linearity* score but a low *line critic* score. The designer has requested that the level fit the line shown in pink (which remains flat across the whole level), which the level does not, hence the low line critic score. However, the level does fit well to a line that slants upwards, hence the high linearity score..... 191

Figure 55. **Tanagra lenience example.** Two levels generated by Tanagra, with the same linearity score but different lenience scores. The top level has a lenience score of 0.61, the bottom level has a lenience score of 0.12. 192

Figure 56. **Three rhythm groups created by Launchpad**, A, B, and C (top to bottom). 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. 194

Figure 57. **Launchpad expressive range graph.** The expressive range for Launchpad when all input parameters are weighted equally. The gradient bar on the left shows the scale for all graphs in this dissertation: deep blue corresponds to a single level, bright red corresponds to over 75 levels. 196

Figure 58. **Launchpad expressive range graph, results from disabling geometry repetition.** Launchpad's expressive range when the options for increasing previously chosen geometry and repeating rhythm groups are turned off. 197

Figure 59. **Launchpad's expressive range graphs for each combination of rhythm parameters.** Graphs are arranged in clusters of four by row and column. Left to right: all rhythm lengths, length 5, length 10, length 15, length 20. Top to bottom: all rhythm types, regular type, random type, swing type. Within each cluster of four, clockwise from top left: all density, low density, medium density, high density. 199

Figure 60. **Launchpad expressive range with varying rhythm types.** The expressive ranges resulting from varying only the **type** of rhythm used by Launchpad and leaving the rest of the parameters equally weighted. 200

Figure 61. **Examples of Launchpad expressive range leniency biasing (graphs).** Expressive range graphs for Launchpad where the rhythm type is swing and the rhythm density is low. The left shows rhythm lengths of 5 seconds long, the right shows rhythm lengths of 10 seconds long. The graph for length 5 medium density swing rhythms looks similar to the graph on the left. 201

Figure 62. **Examples of Launchpad expressive range leniency biasing (levels).** Two Launchpad levels generated with parameters length=5, type=swing, density=low. The top level has low leniency due to the large number of gaps spread throughout the level, the bottom level has high leniency, as there are fewer ways for the player to come to harm.. 201

Figure 63. **Examples of Launchpad expressive range linearity biasing (graphs).** Expressive range graphs for length 10 and 15, low and high density, regular rhythms. The generator is biased towards creating more linear levels, and the non-linear levels it creates have a narrower leniency range. 202

Figure 64. **Impact of removing geometry repetition on Launchpad's linearity-based expressive range.** The expressive range of Launchpad with parameters length=15, type=regular, and density=high, but without any probability for repeating geometry. 203

Figure 65. **Two Launchpad 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. 204

Figure 66. **Two screenshots of the rhythm group cluster visualization tool, showing two different highlight clusters.** The cluster in the upper screenshot shows a cluster of rhythm groups consisting of platforms with gaps in between them that tend to dip downwards in the middle. The cluster in the lower screenshot is of rhythm groups that form a descending staircase punctuated by stompers. 206

Figure 67. **Tanagra expressive range graph.** The expressive range graph for Tanagra with its default beat timeline and no user constraints. 207

Figure 68. **Tanagra: a level with a linearity score of 0 and a leniency score of 0.3.** This is one of the most common kinds of levels that Tanagra creates. 208

Figure 69. **Tanagra: a level with a linearity score of 0.75 and leniency score of 0.5.**..... 208

Figure 70. **Tanagra: a level with a linearity score of 0.9 and leniency score of 0.1.**..... 209

Figure 71. **Tanagra's expressive range, changes due to a level editing scenario.** Tanagra's expressive range in its initial state, then after each of the 5 phases of editing described in this use scenario. At each stage of editing, the designer is shaping Tanagra's expressive range in a predictable way, while Tanagra is still showing that it is capable of producing a variety of levels within the parameters associated with the designer's edits. 211

Figure 72. **An example of a level produced by Tanagra after the first phase of editing.** There are more, and shorter, beats in the middle of the level. 212

Figure 73. **An example of a level produced by Tanagra after the second phase of editing.** There are now long beats at the beginning and end of the level, forcing flatter geometry there than elsewhere, thus removing the likelihood of making the most linear levels. 212

Figure 74. **An example of a level produced by Tanagra after the third phase of editing.** The designer sets preferences for geometry in phase 3 of editing that tend to increase the appearance of stompers and enemies in the level. 213

Figure 75. **An example of a level produced by Tanagra after the fourth phase of editing.** The designer changes the pacing model in the level and the geometry preferences so that short beats should contain gaps and long beats should not. 213

Figure 76. **An example of a level produced by Tanagra after the final phase of editing.** In the final phase of level editing, the designer pins the end of the first beat to be higher up than the middle of the level, and the beginning of the final beat to be the middle y position between the start and middle of the level. 214

LIST OF TABLES

Table 1. **Launchpad parameters.** The parameters for Launchpad that a user can manipulate. 72

Table 2. **Example constraints on beats and geometry in a partial level.** A representative sample of the constraints imposed on the example level structure shown in Figure 30..... 117

Table 3. **Difficulty tiers in *Endless Web*.** Each compositional parameter in Launchpad is linked to a challenge type and an associated fear. There are three different difficulty tiers for each type of challenge. Each challenge is color-coded: enemies are red, platform hazards are orange, springs are blue, gaps are green, moving platforms are yellow, and stompers are purple. 148

Table 4. **The distribution of dreamer and power-up placement across Launchpad's generative space.** Numbers refer to the different challenge tiers. Power-ups are placed between main tier levels. Cells are color-coded by the color associated with each challenge type; darker colors indicate a more difficult version of the challenge. Numbers following dashes denote the number of spaces above a tier; these correspond to narrow lines in the web visualization of the generative space. 150

ABSTRACT

Expressive Design Tools: Procedural Content Generation for Game Designers

Gillian Margaret Smith

Games are shaped by the tools we use to make them and our ability to model the concepts they address. Vast improvements in computer graphics technology, processing power, storage capacity, and physics simulations have driven game design for the past forty years, leading to beautiful, spacious, detailed, and highly immersive worlds supporting games that are, for the most part, fundamentally about movement, collision, and other physics-based concepts. Designers use increasingly complex tools that support the creation of these worlds. The games industry is undoubtedly thriving on these technologies, but what comes next? The tools that game designers use today are highly complex—too complex for a growing population of novice designers, and offering little support for reasoning about gameplay—and there are entire genres of games that cannot be made using the tools available to us now.

This dissertation presents the use of procedural content generation to create *expressive design tools*: content generators that are accessible to designers, supporting the creation of new kinds of design tools and enabling the exploration of a new genre of game involving the deep integration of procedural content generation into game mechanics and aesthetics. The first of these tools is Tanagra, the first ever AI-assisted level design tool that supports a designer creating levels for 2D platforming games. Tanagra guarantees that levels created in the tool are playable, and provides the designer with the ability to modify generated levels

and directly control level pacing. The second tool is Launchpad, which supports a designer controlling both component and pacing features of generated levels; its companion game *Endless Web* uses the generator to create an infinite world for players to explore and alter through their choices. *Endless Web* is one of a handful of games in a new genre enabled by content generation: PCG-based games. Finally, this dissertation presents a novel method for understanding, visualizing, and comparing a generator's expressive range, thus allowing designers to understand the implications of decisions they will make during the design process.

ACKNOWLEDGMENTS

As a child, I dreamed of growing up to be an inventor; I feel incredibly fortunate and grateful to be fulfilling my dream. I could not have written this dissertation or performed any of the work leading up to it without the support of my family and friends, or the many happy accidents that have brought me to where I am today.

Firstly, I thank my parents and brothers for their support, encouragement, and steadfast belief in my ability to do whatever I put my mind to. I should also thank my brothers for the countless hours of playing competitive *Sonic the Hedgehog 2* growing up—I began the research contained in this dissertation a long time ago!

Computer science wasn't an obvious career path for me, and I have many people to thank for me making it as far as grad school: Terry and Daniel Mayfield, for introducing me to computing through playing in tank simulators and dismantling hard drives; Doug Bigelow, for encouraging me to stick with it in the face of many doubts; Tom Hutchinson, who encouraged me to pursue a PhD; and Greg Humphreys, for showing me that computing is an artistic medium and suggesting that I apply to UC Santa Cruz.

I feel incredibly fortunate to have come to UC Santa Cruz at the same time as the Center for Games and Playable Media was being formed. My advisor, Jim Whitehead, has been a constant source of good-humored guidance, support, and encouragement, without whom none of this would be possible. I am also grateful to have worked with Michael Mateas and Noah Wardrip-Fruin in the Expressive Intelligence Studio; their feedback on my work

throughout graduate school has been invaluable. Michael Young's suggestions and feedback on my work, from thesis proposal to defense, have been extremely helpful.

I have been lucky to work with many wonderful collaborators: Mee Cha and Mike Treanor (Launchpad); Martin Jennings-Teats (Polymorph); and Alexei Othenin-Girard, Elaine Gan, Rob Segura, Rob Giusti, Jameka March, Joshua Ray, Masami Kiyono, Vencenza Surprise, Ari Burnham, and Umi Hoshijima (Endless Web). I am also grateful to Anne Sullivan, Adam Smith, Mark Nelson, Josh McCoy, Mirjam Eladhari, and everyone else in the Center for Games and Playable Media for their friendship and influential research discussions.

Greg Levin, Caitlin Sadowski, Jaeheon Yi, Ian Pye, Adam Smith, and Dan Damiani have provided welcome distractions with board game nights (or full days) and evenings watching Babylon 5 together. I am grateful to my crafting friends who have provided me with an escape from science and engineering, and helped me find a way to express myself non-digitally: Tina Michalik, Julia Williams, Dana Harris, Susy Trower, Nadine Schaeffer, and the South Bay Area Modern Quilt Guild have brought much needed balance into my life.

There are so many things to thank Anne Sullivan for I barely know where to begin. Anne has been my partner in goofiness, research, teaching, travel, crafting endeavors, and countless other adventures throughout grad school—from serious research debate to driving hours away for good waffles. I could not ask for a kinder or funnier best friend and doppelganger.

Finally, I thank David Bigelow for his constant love, companionship, encouragement, and tolerance of both my impulsive nature and my tendency to over-work. He has brought me happiness every day.

CHAPTER 1

INTRODUCTION

Game design has historically been taken advantage of the development of new technologies. The first game specifically designed for a computer, *Tennis for Two* (Higinbotham 1958)[*], was created to showcase the technical capabilities of the Systron-Donner analog computer and designed to fit within the constraints of the computer's architecture and the oscilloscope's display capabilities (Nyitray 2011). Technology both constrains and drives game designs; Jesse Schell notes,

> *The technology is essentially the medium in which the aesthetics take place, in which the mechanics will occur, and through which the story will be told.*
>
> (Schell 2008, pp.42–43)

The hardware on which games run has forced constraints on the kinds of games that can be built (Montfort & Bogost 2009), and changes in game controllers have enabled new kinds of games (Sung 2011).

The primary technology advancing digital games in the last 40 years has been computer graphics. From monochromatic screens, to color monitors, to high definition displays, and from "racing the beam" (Montfort & Bogost 2009), to sprite-based 2D graphics, to 3D real-

[*] *Nim* (Bennett 1951; Donovan 2010, pp.5–6) and *OXO* (Douglas 1952) both preceded *Tennis for Two*. However, these games are both exact clones of existing games played with pen and paper. *Tennis for Two* was the first game that could not exist without the computer it was created for.

time rendering, the main goal has been to create more immersive, visually pleasing worlds for players to inhabit.

Juul describes games as "half-real", an amalgamation of rules and fiction[*] (Juul 2005). Advances in computer graphics have driven improvements and new opportunities for the fiction of game worlds and properties of their fictional design. Consider, for example, the game *Wizardry* (Greenberg & Woodhead 1981), in comparison with the game *Dragon Age* (BioWare 2009). 28 years of advances in graphics technology means that the players are exploring a larger, more detailed, more complex, and more immersive world when playing *Dragon Age* than *Wizardry* (Figure 1). However, the "rules" side of many of these games has gone largely unchanged.

Hunicke et al.'s MDA framework (mechanics, dynamics, and aesthetics) provides a vocabulary to explain Juul's "rules" side of game design (Hunicke et al. 2004). The mechanics of both *Wizardry* and *Dragon Age* are extremely similar; both games have a party-based system made up of characters that vary in appearance, gender, race, profession, and skills. The party adventures through a scripted world, fighting monsters and collecting treasure. The aesthetics of both games, where "aesthetics" refers to the reason why the game is compelling, are also similar; in the MDA taxonomy of game aesthetics, which Costikyan describes in further detail (Costikyan 2006), the games exhibit fantasy, narrative, challenge, and discovery. These aesthetics are reached through the interactions between the player

[*] Juul cautions, as do I, that fiction here should not be taken to mean "storytelling", which is more about the story events presented to and caused by a player's behavior, and therefore intertwined with the "rules" side of the game.

Figure 1. **Screenshots from *Wizardry* and *Dragon Age: Origins*.** Advances in computer graphics technology mean that the player is exploring a more detailed and immersive world in *Dragon Age*, but the gameplay of the two games is very similar.

and the rules—the dynamics of the system—such as forming combat strategies, learning about an unfolding story, and exploring the space of the game.

The main reason for these similarities in the rules of the game is that the models supporting their mechanics, and therefore the strategies players form around those mechanics and the resulting aesthetics, are not much different. Improvements to processing power, storage capacity, and programming languages mean that is it easier to store and express more

complex worlds and rule systems, but the underlying models are very similar. It is relatively simple to create mathematical models of combat, following the rules set forth by D&D (Gygax 1978) and other pen-and-paper based roleplaying systems, and execute those rules. Technological advancements for the "rules" half of game design have mostly extended the capabilities of existing representations and processes—what Mateas and Wardrip-Fruin call "operational logics"—rather than defining new ones. Computers excel at understanding concepts such as spatial movement, physics and collision detection, and resource management. And while there is still room for a number of new kinds of games within the space defined by current capabilities, there are entire new genres of games that cannot be reached without the creation of new technologies:

> *…the space of possible innovation is not free—it is fundamentally constrained by the operational logics available. This is because operational logics deeply underwrite mechanics and rules … the definition and development of new types of logics presents an important alternative to creating games, commercially or otherwise, that primarily depend on longstanding spatial and resource management logics…*
>
> (Mateas & Wardrip-Fruin 2009)

Part of the future of game design, then, lies in the creation of technologies that can drive the rules aspect of game design as much as computer graphics has driven the fictional.

Another important future for game design relies on broadening participation in the design process. Designing games requires high degrees of skill in a variety of fields, and the tools

used by game developers rely on specialized knowledge and tend to have a high learning curve. The democratization of game development has already begun, with the creation of tools such as Scratch (Resnick et al. 2009), Alice (Cooper et al. 2000; Kelleher et al. 2007), GameMaker (Overmars 1999), and Kodu (MacLaurin 2009) that reduce the complexity of programming interactive media, allowing children and novices to create their own games. Games such as *Little Big Planet* (Media Molecule 2008) and *Spore* (Maxis 2008a) encourage players to create their own content and share their creations with the online community. Players of these games have collectively created millions of creatures and levels[*]. The same technologies that can drive new game rules and fictions can also create new tools to help with the *design* process for games as much as the *development* process.

Consider, for example, *Spore Creature Creator* (Maxis 2008b), the tool that allows players to easily and playfully design 3D models of creatures that will inhabit the *Spore* game world and that can be shared with the rest of the game's community online. The procedural animation and texturing technology underlying *Creature Creator* is vital to its function; without it, it would be far too difficult for players to create professional-quality models that can walk, dance, display simple emotions, and inhabit the game's fiction. In contrast, *Little Big Planet*'s level editing tools provide support for the look-and-feel of levels but have no underlying "rules"-side design support. This results in a bifurcation of level designers; advanced players can push the system beyond its original intended purpose, creating levels that the designers likely never expected, such as the "Little Big Computer" (Anon 2008). The

[*] On January 17, 2012, Media Molecule announced that six million *Little Big Planet* levels have been created (Crawley 2012). As of April 29, 2012, the *Sporepedia* contains over 172 million *Spore* creatures, vehicles, buildings, and adventures (Maxis n.d.).

majority of players, however, are novice designers who are given no support for creating levels that produce the same qualities of player experience as professionally designed levels.

The need for more game design-oriented tools is not restricted to just novice designers. Improvements in computer graphics technology have driven the creation of complex, specialized tools for creating 3D models and animations (Autodesk 1990; Autodesk 1998), collaborative world building (Simutronics Corp. & Idea Fabrik Plc. 2011), and virtual cities (Epic Games 2010; Procedural Inc. 2010; Smelik et al. 2011a). There are far fewer tools that allow designers to explore aspects of player experience, however, and none that have reached the maturity of the tools used in professional game design[*]. There has been a call for creating similarly supportive and advanced tools for the rules and content aspects of game design, to enable professional designers to focus on a player's core experience by being able to easily prototype and view the impact of different design decisions (Isla 2009; Satchell 2009). There is a clear need for building technologies to support gameplay-aware tools that can assist a designer during the design process.

1 PROCEDURAL CONTENT GENERATION

Obviously, there are many potential directions to explore for building intelligent tools for game design and methods for enabling new game mechanics. A core technology that can help in both of these areas, however, is **procedural content generation**, or the programmatic creation of game content. Similar to *Creature Creator*'s use of procedural techniques to support players creating complex models, content generation can serve a role

[*] Smith et al.'s BIPED tool for producing hypothetical playtraces for a given design (Smith et al. 2009b) and Dormans's Machinations framework are two examples of early work in building such tools (Dormans 2012).

as a companion game designer in a tool that helps designers safely experiment with different options for content by prototyping their ideas within a structured editor. New game mechanics can arise when content generation is used as a tool in the game designer's toolbox—the generator can act as an on-demand game designer who can be called on to create environments that morph and adapt to a player's choices or play-style.

1.1 GAME DESIGN

In order to discuss how procedural content generation can play a role in the game design process, it is first important to explain what exactly is meant by "game design". Fullerton describes the role of a game designer:

> *The game designer envisions how a game will work during play. She creates the objectives, rules, and procedures, thinks up the dramatic premise and gives it life, and is responsible for planning everything necessary to create a compelling player experience … the game designer plans the structural elements of a system that, when set in motion by the players, creates the interactive experience.*
>
> (Fullerton 2008, p.2)

The idea of game design as creating a player experience also resonates with other designers. Brathwaite and Schreiber similarly describe game design as the creation of content and rules, but go on to state:

> Good *game design is the process of creating goals that a player feels motivated to reach and rules that a player must follow as he makes meaningful decisions in pursuit of those*

> *goals … Good game design is player-centric. That means that above all else, the player and her desires are truly considered.*
>
> (Brathwaite & Schreiber 2008, p.2)

Schell also echoes this statement, bluntly reminding us that "the game is not the experience … the game enables the experience, but it *is not the experience*" (Schell 2008, p.10).

Level design, the specific aspect of game design that this dissertation addresses, is crucial to good game design. Kremers argues that "level design is the application of game design" (Kremers 2009, p.26); similarly, Byrne describes levels as "a container for gameplay" (Byrne 2004, p.137). Levels provide the player with a space to explore, learning and mastering the game's mechanics and discovering the underlying story. In Juul's definition of games as rules and fiction, levels play both roles:

> *The level design of a game can present a fictional world* and *determine what players can and cannot do at the same time. In this way, space in games can work as a combination of rules and fiction.*
>
> (Juul 2005, p.163)

Game design is therefore concerned with making decisions about what the rules of the game are, what different kinds of elements should be used, and the story and overall progression of the game if applicable, all towards building a particular kind of player experience. Level design is the process of constructing specific spaces in which the player interacts with the rules of the game. The processes of game and level design are both highly

iterative (Zimmerman 2008); it is impossible to understand the impact of changing the rules of a game or the layout of a level without testing it with real—or, more recently, simulated (Salge et al. 2008; Smith et al. 2009a)—players.

1.2 Procedural Content Generation for Game Design

At its core, procedural content generation is simply the automatic generation of content. There is a large history of work in computer graphics to procedurally generate textures, terrain, and natural phenomena such as water, smoke, and fire (Ebert 2003; Perlin 1985). These have largely been advances in computer graphics as applied to creating immersive virtual environments; this work has been useful for advancing the fiction half of game design, but has rarely addressed the rules half. **Procedural content generation for game design** focuses on the programmatic creation of content that is closely tied to a particular game's design and desired player experience. Examples of procedural content generation for game design (hereafter called simply "PCG") include the construction of interactive environments, automatically generated weapons that the player will use in different ways throughout the game (Hastings et al. 2009), or a generated story that adapts to actions taken by the player (Thue et al. 2008).

PCG can only be a useful tool for designers if it is sufficiently controllable and expressive. Content generators can excel at rapidly creating a huge quantity of content, but in order for the generator to be useful in design it must also be capable of creating a large variety in the style of that content, and it should be able to be meaningfully directed by the designer. The goals and requirements for this kind of PCG system are quite different from prior work in procedural content generation, which has largely focused on the design of systems that can

create realistic looking natural effects or replicate the efforts of a human designer, rather than interact with one.

PCG systems have the potential to act as on-demand game designers, rapidly creating content that meets the needs of designers and/or players and reasoning about the quality of that content. For this future to be realized, content generators must be as flexible as possible. There are two major abilities a generator needs to achieve this flexibility:

1) Understanding a game's design and the experience being built for the player, and
2) Be capable of receiving *direction* from its user.

1.2.1 Reasoning about Game Design

To be an on-demand game designer, a generator must be capable of *design*, reasoning about not only the configuration of components but also how that configuration changes the player's experience. In the context of level design, this means the generator needs to understand crucial level design concepts such as game pacing or how the level fits in with the game's story. Without design understanding, the generator will not be able to communicate with its user in a meaningful way, and is reduced to merely randomly recombining the building blocks that are specified by the generator's designer.

Most current PCG techniques place the majority of this understanding in the knowledge representation layer, frequently embedded in the building blocks used by the generator. For example, the generation algorithm used in *Diablo III* (Blizzard North 1997) takes as input a set of modular level "templates" – abstract layouts of areas that have a uniform size. These templates are placed together in a valid configuration, so that it's possible to move between

templates in the level. The templates are then replaced by one of many different fully instantiated, human-authored level segments (Regier & Gresko 2009). The algorithm itself does not understand the content being pieced together; the majority of the *design* in the process is done by human designers before the algorithm is even run, and with little concern for how the algorithm works.

In designing a content generator for game design, design understanding must be incorporated into the generator. The generator should understand not only what pieces can be put together, but how they should be connected and what impact the combination of pieces has on player experience (see Chapter 2, Section 1.1.1 for further discussion). The *Diablo III* generator mentioned above does not have any understanding of how its level chunks should be placed together, and so can place no guarantees on the qualities of its output. It might create levels that have an undesirable number of dead ends, for example. The need for game design awareness in the generator becomes clear in the case of procedural level generation in *Spore* (Maxis 2008a): in order to ensure that game design constraints such as terrain navigability were obeyed, the design team had to generate and test a large variety of generated levels to create a set of "approved" seeds to the content generator. This provided designers with the ability to blacklist certain generated content that was unacceptable for release, and exert control over gameplay aspects of generated content. These seeds were the only ones allowed to be used in the game released to the

public[*]. An understanding of game design built into the generator would alleviate this problem.

A content generator can only be as strong and flexible as the building blocks it has to work with. By placing game design knowledge into the generator, it is possible to increase the granularity of the input and have the generator work with smaller, more atomic units of game design. In the context of creating expressive design tools, smaller units mean the computer is working with the same granularity of buildings blocks as the human designer, allowing for communication about the content being created in both the small scale building blocks and larger scale concerns about how those blocks fit together.

1.2.2 Designing for User Control

The second crucial aspect of creating a generator that can act as a game designer is providing a means for a human user to communicate with the generator and exert some amount of control over the content it can create. There are two different kinds of control that a PCG system must permit: control over the composition of the content, and over the intended player experience.

While the generator should be able to create a wide variety of content (see Chapter 7 for a description of how to both measure and visualize this variety), there must be a way for either a designer or a player to tailor this content. The type and extent of control required is largely determined by the context in which it will be used. In a tool for assisting designers with creating a piece of content for a game, the designer must be able to very tightly control the generated content; the generator and human designer should be able to communicate

[*] Personal communication with Kate Compton, former technical artist at EA/Maxis for *Spore*.

about both abstract game design concerns and the precise configuration of content being created. A game that deeply incorporates procedural content generation into its mechanics, on the other hand, might require less precise control over the placement of the generated content, but require deeper control over more abstract aspects of the content relevant to the game's design.

Many generators do not offer any sort of gameplay control. Any impact on the play experience is considered emergent behavior, and cannot be specifically designed for. Furthermore, control over the appearance of the content requires modifying the generator code itself. PCG for game design requires systems to make content control accessible to non-technical designers, either as a set of input parameters or through direct or indirect manipulation of an artifact made by the generator. The generator should be able to be *directed* to act in a certain way or towards a certain goal.

2 2D Platformer Games

The work described in this dissertation focuses on the domain of 2D platformer games. These are games characterized by a 2D environment, viewed from the side[*], which the player navigates mainly by running along platforms, jumping over obstacles, and collecting coins or other special items. The core mechanic of a 2D platformer is jumping. The primary goal of the player is typically to successfully navigate a level from one end to another; secondary goals common to the genre include collecting as many coins as possible (Nintendo EAD 1990; Sonic Team 1991), seeking out hidden areas (Nintendo EAD 2006a;

[*] Most of these are side-scrolling games, where the world continues in front of and behind the avatar, and moves continuously as the avatar moves through the world. Early 2D platformers, however, would have either a single screen or swap between screens when the player reached the edge of one.

Figure 2. **Two examples of 2D platformer games.** Left: a portion of the Yoshi's Island 2 map from *Super Mario World*. Right: A portion of the Marble Zone I map from *Sonic the Hedgehog*. These two games have almost identical core mechanics, yet very different styles of level design.

Rareware 1995), or solving small spatial puzzles (Armor Games 2008; Frontier Developments 2008). The specific kinds of platformer game studied in this work are those which focus on players overcoming dexterity challenges.

Platformers are well-suited for research in procedural level design due to their relatively simple and well-understood rules but emergently complex level designs: despite the simplicity of their mechanics, there is large variety in levels within the genre (Figure 2 shows excerpts from levels for *Super Mario World* (Nintendo EAD 1990) and *Sonic the Hedgehog* (Sonic Team 1991)[*]). For example, the popular game *Sonic the Hedgehog 2* (Sonic Team 1992) has only three movement mechanics—running, jumping, and spin dashing—yet a massive variety in level elements and configurations.

3 Research Contributions

The primary motivation of my work is a desire to create procedural content generators that can drive advances to the "rules" half of game design as much as computer graphics has

[*] These level excerpts are larger than what is seen by the player during gameplay. They are cropped from the original maps, downloaded from http://www.vgmaps.com, accessed April 29, 2012.

influenced the “fiction”. Specifically, there are three main research questions that prompted the work described in this dissertation:

1. How can we design content generators that support a new paradigm for design?
2. How does the incorporation of procedural content enable the creation of a new game genre?
3. How can we evaluate and understand a content generator’s expressive capabilities and communicate them to designers?

Answering these questions requires the creation of different kinds of content generators from those that have come before; I call these new generators **expressive design tools**. Such tools are defined as:

> **Generators imbued with an understanding of a game’s design that are sufficiently controllable and expressive for use in the design process.**

Unpacking this definition reveals four main aspects expressive design tools: design understanding, control, expressivity, and intersecting the design process. This dissertation describes contributions I made in each of these four areas.

3.1 Design Understanding

In Section 1.2.1 I described the need for content generators to understand the kind of game for which they are creating content. Such an understanding requires a deep analysis of similar games and a formalization of both the structure of content and the way in which the

player will experience it. This dissertation contributes a novel analysis of level structure and player experience in 2D platforming game, which provides an understanding of how levels can be broken down into atomic components and reconstructed using a rhythmic representation for level pacing.

3.2 Control

Expressive design tools must be controllable by a designer who is often not familiar with computer programming. Section 1.2.2 discusses the requirement that generators must permit control over both composition and player experience. This discussion is continued in Chapter 2, Section 1, in which I define a new taxonomy for procedural content generation. It is the first taxonomy of PCG systems to focus on the control available to a designer using the system, both in terms of the type and extent of the control. The two expressive design tools described in this dissertation, Launchpad (Chapter 4) and Tanagra (Chapter 5) also offer novel forms of control to a designer; Launchpad provides parameterized control over gameplay relevant properties of the levels it creates, while Tanagra is the first PCG-driven game design tool that offers a designer direct control over generated content.

3.3 Intersecting the Design Process

There are two main ways in which this dissertation describes the intersection of procedural content generation and the game design process: PCG-assisted design, in which the content generator is assisting a human designer at design-time, and PCG-based game design, in which the content generator is used to create new content at play-time.

3.3.1 PCG-Assisted Design

Engelbart writes of the potential for computers to be used as tools that "augment human intellect" (Engelbart 2003). Our writing has been influenced by the development of technologies from the pencil to the word processor. The incorporation of procedural content generation into a level design tool has the potential to similarly influence and augment a human level designer.

This dissertation describes a new paradigm for design in a content generator can react to changes and requests made by a human designer. Tanagra is a tool that provides intelligent assistance for level designers of 2D platforming games by incorporating PCG into the level design process. It is the first mixed-initiative design tool for games, and the first to provide a designer with direct control over level pacing without needing to manipulate level

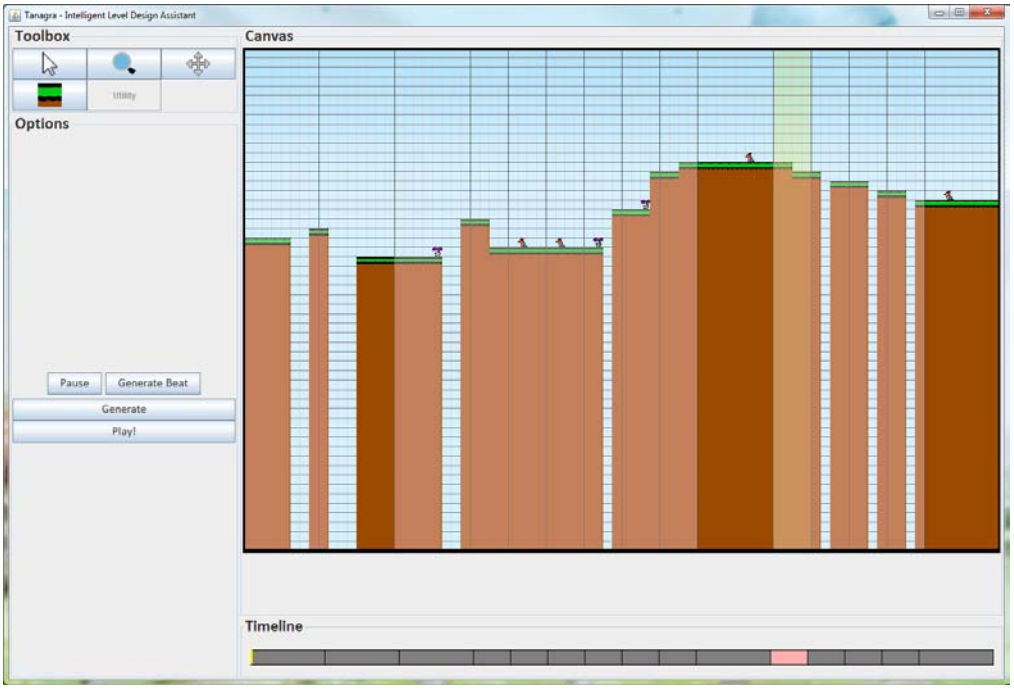

Figure 3. **A level being designed with Tanagra.**

geometry. The content generator plays three major roles in the design process: suggesting content for the designer, allowing the designer to experiment with different level pacing options, and ensuring that all levels made in the tool are valid and playable. A screenshot showing Tanagra is in Figure 3.

3.3.2 PCG-Based Game Design

PCG can act as a tool for game designers to create entirely new kinds of games; this idea is explored in the creation of *Endless Web*, which uses the Launchpad level generator. Launchpad is an expressive design tool that was the precursor to Tanagra, and is the first level generator in the domain of 2D platforming games that provided a designer with parameterized control over qualities of the produced level. Launchpad is also one of a handful of level generators that provides control in a manner that is accessible to a non-technical designer. Launchpad's abilities as a tool for use in designing a game are evaluated through the creation of the game *Endless Web*.

Endless Web is an experimental 2D adventure/platformer that deeply integrates procedural content generation into both its mechanics and aesthetics. It is the first game that has the player directly interact with a content generator to control structural aspects of the world

Figure 4. **Screenshots from *Endless Web*.** These screenshots showi examples of two different configurations of *Endless Web*'s generative space.

she must explore, thus requiring her to strategize around the content generator itself. It is one of very few games that deeply integrates procedural content generation into every aspect of its design. *Endless Web* occupies a new genre of game: the PCG-based game. PCG-based games offer a new direction for game design, in which game worlds are deeply responsive to player's choices. This kind of game has the potential to provide greater agency and a wider variety of player experiences.

In *Endless Web*, the world is procedurally generated as the player moves through the game, shifting according to choices that the player makes. Players must search through the procedural rather than the physical space of the game in order to complete their objectives. The generator is influenced by both the choices made by the player, which control the composition of the world, and by the game designer's decisions on how level pacing should change over the course of the game. Figure 4 shows two different screenshots from Endless Web showing different configurations of the world.

3.4 Expressivity

The variety in the range of content a generator can create, and controllability of that range, defines the expressive range of a generator. Generators that are used in game design should have a high expressive range: a large variety of content that can be created and a clear and predictable way to control that range of content, with any biases in the generator well-exposed and understood. Prior to the work presented in this dissertation, the primary method for evaluating content generators was simply showing hand-picked examples of content created with the generator. I describe the first method for evaluating, visualizing, and understanding a content generator's expressive range; this method allows different

generators to be rigorously evaluated, compared against each other, and analyzed for different use cases.

There are two main research questions that arise when describing the expressive range of a content generator:

1. What metrics should be used to rate expressivity, and
2. How can the expressivity of the generator be visualized and compared?

The expressivity analysis of both Launchpad and Tanagra is performed using two metrics judged important for level design: the **leniency** and **linearity** of generated levels. Leniency refers to how likely the player is to come to harm during the navigation of the space, and linearity is an aesthetic quality of levels that describes how well the level fits to a line in space. Expressive range is visualized using two main techniques: two-dimensional histograms showing the generator's coverage of the space defined by linearity and leniency, and an interactive tool for exploring the similarities and differences in generated content.

4 Dissertation Overview

Chapter 2, *Procedural Content Generation for Game Design*, situates the work described in this dissertation in the context of existing research in the research fields of AI assisted design tools and procedural content generation research, and further lays out the requirements for building expressive PCG tools for game designers.

Chapter 3, *Design Space Analysis*, describes the particular domain within game design that this dissertation addresses–that of 2D platforming games–and describes the motivation for

choosing this domain. It describes a rhythm-based structure for level design derived from the analysis of several representative games from the genre. This representation of rhythm is used in both the Launchpad and Tanagra generators to allow a human designer to direct the pacing of generated levels.

Chapter 4, *Launchpad: Designer-Influenced Level Generation*, describes the first version of the Launchpad procedural level generator, before its modification for use in *Endless Web*. The chapter opens with a discussion of the need for level generators used in games to be controllable, then provides the technical implementation details of a system that supports this kind of control. Launchpad permits a designer to change a number of different parameters describing the desired level pacing and composition before content generation.

Chapter 5, *Tanagra: Mixed-Initiative Level Design*, describes the Tanagra tool. This chapter provides both the motivations and design goals for the Tanagra tool and the technical details of its implementation, inspired by the rhythm-based analysis from Chapter 3. It describes how the integration of reactive planning language, using ABL (Mateas & Stern 2002), and numerical constraint solving, using Choco (Choco Team 2008), can create a generator that can communicate with a designer about both the abstract structure of a level and the concrete placement of level geometry. This chapter also shows a typical use case for the tool, describing the different ways in which designers can interact with it.

Chapter 6, *Endless Web: PCG-Based Game Design*, describes the capability of PCG to enable entirely new kinds of gameplay. It defines the new genre of PCG-based games, as opposed to games that simply use procedural content generation in the service of game mechanics

that are standard to other genres. In this chapter I also describe the design process used in creating *Endless Web*, the challenges encountered during its design, and ways these challenges were overcome. Finally, this chapter describes the technical modifications made to the original version of the Launchpad generator (which inspired the creation of the game) that were necessary to support *Endless Web*'s mechanics and story.

Chapter 7, *Evaluating the Expressive Range of PCG Systems*, defines the concept of a generator's *expressive range* in terms of both its ability to produce a variety of output and its controllability. This chapter first defines two metrics for 2D platforming levels–linearity and leniency–and then describes two techniques for visualizing the expressive range of content generators, and the application of these techniques to both the Launchpad and Tanagra generators.

Finally, in Chapter 8, *Conclusion and Future Work*, I summarize the contributions of this dissertation, and describe future avenues for research in the area of Expressive Design Tools and Procedural Generation for Game Design.

CHAPTER 2

PROCEDURAL CONTENT GENERATION FOR GAME DESIGN

The work described in this dissertation draws from two main areas of work: procedural content generation and AI-assisted design and creativity support tools. This chapter situates the concept of procedural generation for game designers in the context of both of these fields, examining how previous research in procedural content generation has approached issues of controllability and understanding game design, and how AI does and should play a role in design tools.

1 THE DESIGN OF PCG SYSTEMS

AI-supported procedural content generation is a key enabler for the development of creative systems that assist game designers in realizing new kinds of tools and game designs. There have been many different approaches taken to designing algorithms for procedural content generation, in both academic research and industry games; these approaches are well summarized in surveys of the field (Hendrikx et al. 2011; Togelius et al. 2011). Both of these survey articles place existing PCG systems into taxonomies based on the AI techniques used in generation and the role of the generated content in the game. While these are important ways to classify current work, these taxonomies tell us nothing about how useful a PCG system is in the game design process.

PCG systems that are used by game designers—either by playing a role in a design tool or by forming a core part of a particular game's design—must be expressive, accessible to non-

technical designers, and be capable of reasoning about aspects of the game's design. This section reviews the current state of research in procedural content generation as it applies to these requirements, and introduces a new taxonomy for PCG systems, classifying them by the type and extent of control they provide to a a designer.

1.1 Controlling the Generator's Expressivity

In 1984, the game *Elite* was released for the BBC Micro (Braben & Bell 1984). *Elite* was quite different from the other games produced in its era due to the massive galaxies available for the player to explore, despite the technical limitation of only having 22K of memory. Braben describes their desire to exceed these technical limitations:

> *…that meant there were probably only a dozen places we could create, which wasn't what we wanted to do. So what we did was we compressed the world, and this trickery gave us huge galaxies to explore. Initially we had way more than eight galaxies. For instance, we threw away a whole galaxy because it had the word "arse" in it...*
>
> (Boyes 2006)

Galaxies in *Elite* are created by the computer, but stochastically generated. The designers retained the ability to curate the generated content, to the extent that they were able to edit out objectionable content. *Elite* provides the first generator that offers complete and direct control to a designer, although all of the control is placed in the algorithm itself and, while the generator is capable of creating many different galaxies, a single seed number was chosen to ship with the game and all players see the same content. The content generator in *Elite* essentially serves as a form of data compression. The technique of using PCG for

compression is also common in demoscenes. For example, the game *.kkrieger* (.theprodukkt 2004) is a first-person shooter that uses only 96K of disk space; all of the environments, enemies, and textures are specified as algorithms rather than as content that must be loaded from the disk.

The use of PCG in games like *Elite* and *.kkrieger* occupies an extreme position on three scales that can describe a generator's expressiveness: accessibility of control, extent of control, and variability in content. The way in which the content generator is controlled lies entirely within the algorithm's design; it would be impossible for a user unfamiliar with the algorithm to exert control over the kind of content that is generated. The extent of control over the generated content by the algorithm designers is complete (both compositional and experiential); the generated environments are curated by the game designers and the generators are very specialized towards creating a particular instance of content. But this complete control comes at the expense of variability in generated content; the generators are designed to create one particular instance of content.[*] Procedural content generation for game design requires an ability to curate or control content that both *Elite* and *.kkrieger* exhibit, but also a high level of accessibility for that control and a great variety in the generated content.

1.1.1 Accessibility

Procedural content generation for game designers requires an accessible means for controlling the generator. Many PCG systems use highly specialized algorithms to create

[*] It is worth noting that *Elite*'s generator is potentially capable of creating other content, but the seed used for the content generator is fixed. It is thus impossible to know the true variability present in *Elite*'s generator.

content; any control over the generated content lies in the algorithm, and any attempt to control the kind of content the generator can create requires the extensive modification of source code. This form of control is entirely inaccessible to a non-technical designer. Gingold describes the challenges in providing this kind of control:

> *The danger of moving too much representational work to the programmer/machine is a snarled world; a medium whose complexity is unintelligible and, as a result, implastic. Keeping representation locked up entirely in an artist's static two dimensional images, video, and models, on the other hand, is also untenable, as inert worlds result. Too many dead bits require Frankensteinien acts of computation to gain procedural life.*
>
> (Gingold 2003, p.61)

What is needed in PCG for game design is a form of control that allows a non-technical user accessible control over both the representational and computational aspects of the generator. There are four main ways that a generator can be controlled in a more accessible manner than altering the algorithm itself: through specifying new building blocks, by altering parameters that map directly to generated content, and by altering the shape of a generator's expressive range through either the indirect or direct manipulation of generated content.

Knowledge Representation. All content generators must begin with building blocks. These may be mathematical equations (Hastings et al. 2009; Secretan et al. 2011), human-authored chunks of content (Blizzard North 1997; Mawhorter & Mateas 2010; Persson 2008;

Yu 2009), design patterns (Dahlskog & Togelius 2012; Smith et al. 2011f), or fine-grained grammar production rules (Dormans 2010; Smith et al. 2008a; Smith et al. 2011b; Müller et al. 2006).

A content generator that is designed to be modular with respect to its building blocks allows a less technical designer to specify a library of potential content that can be recombined by the generator. While not necessarily accessible and intuitive for an entirely non-technical designer, one example of such a system is Dormans's mission/space level generator for action RPGs (Dormans 2010), in which a designer can specify the production rules used in the grammar using a specially created tool. Another example is Mawhorter & Mateas's occupancy-regulated extension system (Mawhorter & Mateas 2010) for procedurally generating Infinite Mario Bros. levels, in which arbitrarily sized level components can be specified in an ASCII representation and then recombined in a number of ways by the generator. Carefully constructing the building blocks that a generator uses can have a big impact on the kind of content it creates; however, it can be hard to fully understand the implications of altering these building blocks on the generated content. Guaranteeing sufficient variety, such that the player doesn't notice repeated pattern of geometry, requires a combination of a high authoring burden and an intelligent method for recombining the patterns.

One direction that holds a great deal of promise in controlling PCG systems via knowledge representation is design patterns. Design patterns in the context of games research typically describe either a solution to a design problem or a common design idiom along with how it can vary and what gameplay it affords (Bjork & Holopainen 2004; Dahlskog & Togelius 2012;

Hullett & Whitehead 2010; Smith et al. 2011d). For example, Hullett's design patterns for first-person shooter games describe common features in level design and how they impact the player's experience in terms of tension and pacing. Dahlskog and Togelius's patterns describe how certain combinations of geometry in platformer levels can fulfill either a designer's needs (e.g. a staircase can move from a low point in a level to a high point) or a player's (e.g. several enemies in a row provide an appropriate amount of challenge when the player has mastered killing a single enemy). Good design patterns hold more information than simply permissible configurations of geometry, and thus hold the potential to enable controllable generators that provide a designer with control over many different aspects of player experience. However, design patterns and their impact on player experience are not yet formally specified enough to be reasoned about by a computer.

Parameterized Generators. A parameterized content generator allows a user to tune its output by manipulating input parameters that have a clear and predictable result for the output. The world builder in *Civilization IV* is a good example of this (Firaxis Games 2005); it provides options for gameplay-relevant properties of maps including the size of the map, type of water coverage, size of landmasses, and resource distribution. Another example of a parameterized generator is Hullett and Mateas's collapsed structure emergency rescue scenario generator (Hullett & Mateas 2009), in which a designer specifies goals for the scenario such as a house with partially collapsed walls in particular locations. Launchpad was designed as a parameterized generator. Manipulating appropriately defined parameters can be an accessible means for controlling a generator, and can provide the designer with a

method for controlling not only what building blocks the generator uses but how the generator works to fit those blocks together.

Indirect Manipulation. A common technique used in procedural content generation is genetic algorithms; new generations of content are "bred" according to a specified evaluation function that determines the fitness of generated content and optimizes for a certain aspect of content. Evolutionary PCG systems will sometimes use a human as that evaluation function, on the theory that a human can identify desirable content far better than another algorithm could[*]. This is an example of controlling the generator through indirect manipulation of content; the user typically cannot directly edit content created by the generator, but can influence the next choices the generator takes towards a final solution. Evolutionary systems that support this form of indirect manipulation include the art generation programs Electric Sheep (Draves 1999) and Picbreeder (Secretan et al. 2011), which both use humans to select the next phase of evolution.

Microsoft's Songsmith (Simon et al. 2008) is an example of a non-evolutionary PCG system that is controlled through a blend of parameters and indirect manipulation; users indirectly alter a generated harmony line when they change the melody—the user is not directly manipulating the harmony line itself—and also can alter parameters for the overall song to describe the style and key of the accompaniment. Indirect manipulation is well-suited to exploratory and/or collaborative creativity, in which the final results (if there are any) of the design process are not necessarily as important to the users as the process of exploring

[*] There are also systems that support a human *specifying* the evaluation function in a parameterized way (Dart et al. 2011); note that this is different than using a human as the evaluation function itself. Parameterized specification of the evaluation function is akin to the human declaring a goal for the generator to achieve; using a human as the evaluation function is a more exploratory process.

potential designs and pushing the system to create new, interesting, and even unexpected content.

Direct Manipulation. The final method for controlling a content generator is by posing constraints on the generator through the direct manipulation of an artifact created by it. SpeedTree (Interactive Data Visualization Inc. 2010) is a tree modeling tool that serves as an example of this form of control; users can draw the basic structure of the tree and the tool will turn it into a realistic-looking model of a tree. This is also the control mechanism used in SketchaWorld (Smelik et al. 2011b), in which users directly manipulate the kind of terrain, lengths of roads, and paths of rivers and in doing so constrain the generator's output. Tanagra allows the user to literally add and remove constraints used in the generator by pinning and unpinning platforms and directly drawing level geometry into the scene. Direct manipulation is perhaps the most accessible form of control for a generator in a design tool, as the user can focus on the particular artifact they want to create and receive rapid feedback on the kind of content the generator can create within the posed constraints.

1.1.2 Extent of Control

Dan Kline has commented on the importance of being able to "direct" a player's experience when using procedurally generated content (Kline & Hetu 2011). Players can see through a lack of structure underlying the decisions taken by a generator (a form of the Eliza effect (Wardrip-Fruin 2009)). It is important that this control be over the generator itself, rather than the particular instances of content the generator creates. For example, generated

worlds for *Spore* are "controlled" by having the designers whitelist certain random seeds[*]. Thus there are a large number of generated worlds available for the player, but there was a great deal of human effort required to ensure that these worlds are appropriate for players. Section 1.1.2 talked about the ways in which a generator's user (who may be a player, a designer, or both) can direct a generator; this section discusses the different layers of control that are typically found in generators, which can be described independently from the accessibility of that control.

Indirect. Some generators provide users with a form of control over the generation process that does not have any clear relationship with the artifact being produced. For example, Martin et al.'s 3D building modeling tool provides the user with sliders to control the next phase of evolution for a set of candidate buildings; the sliders control the strength and likelihood of different mutation operations (Martin et al. 2010). It is not easy to predict how the next generation of buildings will look based on the selected parameters. The weapon generation in *Galactic Arms Race* (Hastings et al. 2009) is another example of indirect control, in two different ways. Firstly, the player does not make a deliberate choice to influence the generator; new weapons are generated for the player based on observations of player behavior, not stated player opinions. Secondly, the evolution of weapons only allows the player to "request" more weapons like the one they are currently using; it is impossible to exert control over specific properties of the weapons.

Compositional. Compositional control describes scenarios where the user is capable of controlling the construction of content, such as specifying which building blocks can be

[*] Personal communication with Kate Compton, former technical artist at EA/Maxis for *Spore*.

placed by the generator and how often they should be used. For example, the user can specify a library of level chunks using occupancy-regulated extension (Mawhorter & Mateas 2010), providing control over different patterns of geometry that will appear in the final output. The generator also easily supports code-level changes for the frequency of different chunks being chosen. Diorama, a level generator for the real-time strategy game *Warzone 2100* (Pumpkin Studios 1999), provides the user with control over properties of the map such as the appearance of canyons and causeways, the number of oil wells per player, and the general shape of the map and appearance of water (Anon n.d.; Smith & Mateas 2011). SketchaWorld allows a user to exert a great deal of control over the composition of worlds, by providing tools for sketching terrain types, drawing rivers and roads, and constructing cities (Smelik et al. 2011a). Launchpad and Tanagra both provide a user with compositional control, Launchpad through changing the frequency parameters of components, and Tanagra by moving platforms around the canvas and setting component preference toggles[*].

Experiential. Content generators that provide a user with experiential control are perhaps the most useful kind. Experiential control refers to being able to directly control an aspect of the desired player experience, rather than entirely relying on the player experience emerging from other aspects of control. For example, Diorama provides additional parameters for the user to control the defensibility of bases, terrain reachability, and general map types which have a large impact on the strategies that can be used during the game. The level generator for Refraction supports specifying the educational goals of a level

[*] Note that this control is only available in newer prototype version of Tanagra, described in Chapter 5, Section 9.

by defining the mathematical expression that the players should be constructing in the game (Smith et al. 2012a), though not in a manner accessible to non-technical designers. Launchpad and Tanagra both provide experiential control in the form of manipulating the pacing of levels.

Guarantees vs. Suggestions. There are some aspects of design that the generator must be able to guarantee, while others can be taken as mere suggestions. For example, level generators must usually be able to guarantee that levels they create are playable[*]. Some generators provide more guarantees than others; for example, the Diorama level generator guarantees that there will be exactly the number of oil wells per player as the user requests, while Launchpad promises only a best-fit to the frequency of components appearing in a level. Tanagra makes some guarantees, such as ensuring that user-pinned platforms stay where they are and beat lengths are exactly as the user specifies, but does not guarantee that preferences regarding geometry patterns in beats will be respected.

1.1.3 VARIABILITY

The most common strategy for evaluating procedural content generators thus far is to show examples of the kinds of content that can be produced. For example, showing different racetracks generated according to personalized fitness functions (Togelius et al. 2007), or weapons that support different play styles (Hastings et al. 2009). While this form of evaluation can provide interesting information about the generator's capabilities, it does

[*] The game *Spelunky* (Yu 2009) provides an interesting counter-example to the need for control and the ability to make guarantees about generated content. Where most content generators require controllability at least to ensure that the generated content is valid, *Spelunky*'s design means that the generator is not required to make guarantees about the playability of its levels. Because the player is given tools such as bombs, ropes, and ladders to navigate and re-shape the level, the level does not need to initially contain a path from start to finish.

not fully capture the range of content that can be created and does not easily support analysis of how this range changes for different fitness functions or generation parameters. It is impossible to judge the quality of a level generator based only on these statistics: a generator that can create tens of thousands of levels in a matter of minutes is useless if many of those levels are effectively identical to each other.

Content generators for use in design should be evaluated according to not only the amount of content they can create but also the styles and variety of content that they can produce. The Empath drama management system used an evaluation method similar to one we use for evaluating Launchpad and Tanagra's expressive range, in which they ranked the story goodness for 5000 generated play-throughs, but this was used for analyzing the accuracy of player models rather than teasing out qualities of the underlying AI system (Sullivan et al. 2009a). Chen et al. also evaluated the authorial leverage of the drama manager used in EMPath, a concept similar to expressive range (Chen et al. 2009). To my knowledge, there has been no research other than that put forth in this dissertation for methods for analyzing the variability in content created by a generator.

1.2 Understanding Game Design and Players

Many content generators do not explicitly incorporate any understanding of a game's design or the anticipated player behavior. The aforementioned Occupancy Regulated Extension system (Mawhorter & Mateas 2010), for example, works by combining human-specified level segments that are annotated with how level segments should be fit together. All of the understanding of level design in that system is encoded in these chunks; there is none built into the algorithm itself. In the use of PCG for creating tools to assist designers, there is a

similar lack of game design knowledge built-in. CityEngine (Procedural Inc. 2010) is frequently used to quickly build large cities and buildings for populating a virtual environment, and SpeedTree (Interactive Data Visualization Inc. 2010) is an industry standard tool for creating trees and other foliage for a game world. However, these content generators affect only Juul's "fiction" side of games (Juul 2005): the visual, decorative properties of the game world such as window and façade placement or tree height and species; any spaces that the player will interact with directly are typically still created entirely by the level designer (Golding 2010).

An exception to this is a deep body of literature on personalizing content creation for an individual player. Work in this area includes personalized race tracks for racing games (Togelius et al. 2007), evolving weapons in a space shooting game (Hastings et al. 2009), and adapting Mario levels by manipulating parameters such as gap width and frequency (Shaker et al. 2010). The Launchpad system described in this dissertation has itself been used in such a game, called *Polymorph* (Jennings-Teats et al. 2010), a platformer with level generation-based dynamic difficulty adjustment.

However, game designers generally do not focus on constructing worlds for individual players; therefore, a content generator to support a game designer should be capable of understanding and constructing levels according to general player behavior[*]. There has been very little work in procedural content generation that addresses controlling and generating aspects of gameplay for the universal player (as defined in a taxonomy of player

[*] Incorporating both a universal player model and individual player models into a design tool is interesting future work, however; there is the potential for tools that can create content for games and then show designers how well-suited that content is for particular player types.

modeling that I contributed to (Smith et al. 2011a)). Recent extensions to the SketchaWorld project (Smelik et al. 2011a) aim to enforce gameplay constraints such as guaranteeing a particular line of sight in an environment created in the tool. Launchpad and Tanagra both use a universal player model—that of the rhythm the player feels with her hands while playing—when creating content in the service of game designers.

2 Procedural Content Generation for 2D Platformer Games

A large amount of work in procedural level generation thus far has focused on 2D platformer games, specifically for games similar to *Super Mario World* in style. The approaches taken to generating levels ranges from ad-hoc constructive methods to evolutionary algorithms to answer set programming. Level generation for platformers is extremely challenging due the physics system's constraints on player movement. Compton and Mateas describe these challenges well when discussing why roguelike generation techniques are not appropriate for platformer games:

> *Rogue-like level generation can make heavy use of relatively unconstrained, random decisions. The playability of the level is ensured by the constraints implicit in the human-designed atomic units employed by the generator … in contrast, the playability of a platformer level is strongly determined by the relationships* between *units … the loosely constrained, random placement of elements that works in dungeon level and terrain generation can easily lead to accidentally unwinnable platformer levels.*
>
> (Compton & Mateas 2006)

The first ever platformer game to use procedural content generation was *Pitfall!*, designed by David Crane and released on the Atari 2600 in 1982 (Crane 1982). Each screen of the 255-screen world is generated from an 8-bit number, with 3 bits determining the object pattern, 3 bits determining the ground pattern, 2 bits determining the tree patterns, and 1 bit (shared with the high bit of the tree pattern) determining the underground pattern (Montfort & Bogost 2009, pp.110–112). Much like *Elite*, however, the content generation in *Pitfall!* is deterministic. Crane painstakingly reviewed pseudo-random number sequences until he found an appropriate initial seed that generated early screens that provided an effective tutorial for the game (Crane 2011).

Vib Ribbon (NanaOn-Sha 1999) is the first platformer-like game to incorporate non-deterministic procedural level generation; each level in the game is generated from player-provided music, and consists of a side-scrolling track filled with obstacles that correspond to the main rhythm of the music. Compton and Mateas's work in level generation for platformers is also inspired by the rhythmic structure of music, and constructs patterns of geometry with repetitions to introduce rhythmic elements. Both of these works were inspirations for the rhythm-based level representation used in this dissertation; both Launchpad and Tanagra create levels that are quite similar to *Vib Ribbon* levels in terms of providing various obstacles to match a generated rhythm, and share characteristics such as a single-path level with no opportunity for the player to make choices.

Infinite Mario Bros. (Persson 2008) is an open-source platformer with procedurally generated levels that is designed to look and feel like *Super Mario World* (Nintendo EAD 1990). Levels are generated from left-to-right using chunks of pre-defined patterns, with the

frequency and number of enemies generated based on the level number so that the further into the game the player is, the more enemies he faces. Launchpad and Tanagra both also use pre-defined patterns, but at a finer granularity and with an underlying beat structure driving their length and position.

The open source release of *Infinite Mario Bros.* inspired the creation of the Mario Level Generation Competition in 2010 (Shaker et al. 2011) which had six entries. The goal in the competition was to create a level generator that was personalized to a particular player, using data gathered about the player (such as number of enemies killed and length of play-time) from a random level created with the base *Infinite Mario Bros*. algorithm. Personalized content generation is a popular sub-field within procedural content generation; noted game designer Raph Koster points out that personalized games are the "holy grail" of game design:

> *The holy grail of game design is to make a game where the challenges are never ending, the skills required are varied, and the difficulty curve is perfect and adjusts itself to exactly our skill level.*[*]
>
> (Koster 2004, p.128)

Shaker et al. create personalized levels by learning a model of the relationship between player-reported emotions (fun, frustration, and challenge) and selected input parameters

[*] Koster continues, *"Someone did this already, though, and it's not always fun. It's called 'life'. Maybe you've played it."* Koster's biting remark on the potentially ill-advised quest for personalized games is important to consider—there is a great deal of work that must be done for understanding the worth of personalized content, what should be measured for understanding player behavior and player desires, how much of the player's desires should be respected (e.g. should a player who doesn't wish to enter combat never be given the opportunity to do so?), and how to incorporate personalized content generation into a compelling playable experience.

for the *Infinite Mario Bros.* level generator (Shaker et al. 2010). The "Hopper" generator, created by Glen Takahashi under my supervision, labels a player with an ability level and one of three different styles—speed runner, enemy-killer, and discoverer—based on observed behavior in the game and uses this player model to alter the probabilities for a pattern-based level generator (Shaker et al. 2011).

Other personalized content generators rely on a human designer tagging the input parameters with difficulty information. Shimizu and Hashiyama built their personalized level generator with the goal of having the player experience flow (according to Csikszentmihalyi's definition (Csíkszentmihályi 1991)) by piecing together different chunks, that were themselves generated using interactive evolutionary computation (Takagi 2001) and tagged with a difficulty level by a human designer, according to an estimation of the player's skills and preferences (Shaker et al. 2011). Mawhorter and Mateas's ORE generator can alter the rules for chunk selection based on how chunks are tagged with difficulty information (by the human designer) and an inferred model of player skill (Mawhorter & Mateas 2010). Baumgarten's personalized level generator uses machine learning techniques to classify players into particular skill level and then pieces together pre-designed, tagged level chunks into a complete level (Shaker et al. 2011).

Interestingly, the top performing level generator in the Mario level generation competition did not take any effort to personalize the generated content, nor did the level generator that came in third place. Weber's winning level generator uses a multi-pass approach to first build a playable base level with gaps in it, then performs several other passes to decorate the level with hills, enemies, coins, pipes, and choice blocks. This results in more visually

interesting levels than those created by other generators, using fewer pre-built level patterns. Sorenson and Pasquier use a combination of evolutionary algorithms and constraint satisfaction to evolve levels and then guarantee their playability (Shaker et al. 2011).

The *Polymorph* game which I contributed to (see Chapter 4, Section 5) uses the Launchpad level generator described in this dissertation to dynamically adjust the difficulty of level by taking generated segments, classifying them by their difficulty according to a model learned from humans playing other level segments and rating their difficulty, and selecting an appropriate difficulty segment based on the player's prior performance in the game (Jennings-Teats et al. 2010). Unlike other research in personalized level generation, *Polymorph* generates new content at runtime in front of the player, rather than creating an entire level tailored to a player's inferred preferences. Adam Saltsman, creator of *Canabalt* (Saltsman 2009), describes this ability as the main reason why PCG is "important":

> *This is I think why procedural generation is so important though; not because it saves on content creation time (although that is nice), not because it saves on memory or download size (also nice), not because it has such good replay value (definitely awesome) … Procedural generation is important because of the potential it has for fluid adjustment and reaction, and I think that is something that we're just barely beginning to explore.*
>
> (Lager 2009)

Except for the ORE approach to level generation, all of the approaches mentioned so far involve generating a single main path through the level. Nygren et al. create levels that support multiple paths by generating graphs where each node is a cell and each edge is a portal (using a similar representation to ours, discussed in Chapter 3), and then creating a grid from that graph where each cell is independently filled with level geometry that is guaranteed to connect at grid boundaries. These levels can be personalized based on inferred user preferences; for example, a user who does not confidently kill enemies will see very few enemies in a level generated for him (Nygren et al. 2011).

There have also been a number of 2D indie platformer games that employ procedural content generation. *Canabalt* (Saltsman 2009) and *Robot Unicorn Attack* ([adult swim games] 2010) are similar games that each piece together large, simple chunks of level geometry in front of the player, who is automatically being moved through the level at an increasing speed. *Canabalt* determines the placement of the next building in the sequence based on the player's movement speed and current building height, aiming for creating levels that are challenging but guaranteed to be possible to complete (Saltsman 2010). These two games use an extremely simple form of content generation; the patterns that the generator chooses from are at the same scale as the player perceives them. Launchpad and Tanagra use both a larger variety and a smaller size of building block to generate levels—patterns that are perceptible to the player emerge from the generator—but are well-suited to the speed run style that emerges from *Canabalt* and *Robot Unicorn Attack*.

Terraria (Spinks 2011) uses PCG to generate a large terrain for players to explore and mine for resources, similar to *Minecraft* (Persson 2011). Because *Terraria* is a mining and crafting

game, it does not need to concern itself with level playability in the same way that other 2D platformer level generators do, and instead needs to be more concerned with an appropriate distribution of resources, difficulty of reaching those resources, and monster spawning. It is an example of a game where PCG is used to enable the player exploring a large, unique space that could not be easily made by a human designer.

Spelunky (Yu 2009) is another game where a viable path through the level does not need to be a first-class concern in the generation algorithm. Because the player can use bombs and ropes to traverse otherwise impassable regions of a level, the generator can somewhat rely on the player to find their own path to the level exit. *Spelunky* uses an interesting blend of human-authored and procedural content; each level in the game is made up of rooms, which are in turn made up of a set of tiles. There are 50 different room layouts, specified as strings of characters, where each character corresponds to either a particular level element *or* the probability of a level element appearing. For example, a room might contain a set layout of blocks and platforms, but also have a flag in certain areas that corresponds to placing a set of random tiles in its place. It is thought that after about 500 playthroughs, a player would be able to quickly identify each room layout when encountered in a generated level. *Spelunky* also fills the generated spaces with traps and monsters, which are placed entirely procedurally based on designer-authored rules governing where monsters are allowed to be placed and how many are allowed to be in each level. These monsters are placed without regard to room boundaries in the level (Kazemi 2009). Launchpad uses a similar approach for decorating a generated level with coins.

With the exception of occupancy-regulated extension and, perhaps, *Spelunky* (which also specifies its rooms as ASCII strings), none of the systems described in this chapter provide a non-technical designer with an ability to influence the content that can be created with the generator. These two approaches both use a knowledge-representation level of control, where a human designer can specify chunks that are recombined in the generator using ASCII strings. None of the systems provide the kind of direct experiential control that both Launchpad and Tanagra can provide for level pacing.

3 AI IN TOOLS FOR DESIGN AND CREATIVITY

In addition to research in procedural content generation, the work presented in this dissertation (the Tanagra tool, in particular) is also greatly influenced by prior work in creativity support tools and computer-aided design.

A creativity support tool is one that helps a human achieve her creative potential, by "empower[ing] users to be more productive and more innovative" (Shneiderman et al. 2005). Some of these tools may exhibit creativity themselves, though many are built to intersect a particular phase of the creative design process. Lubart describes four major roles that a computer can play in creativity support tools: nanny, pen-pal, coach, and colleague (Lubart 2005). The nanny role describes tools for helping creators plan their activities, store ideas, or build agendas; tools that serve as pen-pals enable collaboration with other human designers, either in the same location or across time and space. The two roles that this dissertation touches on are the last two: the coach and the colleague. A coach is a specialized expert who can provide guidance to a human designer, and the colleague role describes a tool that can contribute to the solution or the creative work. Both these roles

require an artificial intelligence be built into the tool, to be able to understand the work the user is performing and assist when necessary.

Design activities consist of finding solutions to ill-defined problems, and the design process consists of refining both the problem and the solution in concert with each other.

> *Designers recognize that problems and solutions in design are closely interwoven, that 'the solution' is not always a straightforward answer to 'the problem'. A solution may be something that not only the client, but also the designer 'never dreamed he wanted'.*
>
> (Cross 2011, p.10)

Licklider's suggestion of a future of computing consisting of *Man-Computer Symbiosis* describes the potential for computers to participate in "formulative and real-time thinking", allowing the human to test potential solutions iteratively:

> *…many problems that can be thought through in advance are very difficult to solve in advance. They would be easier to solve, and they could be solved faster, through an intuitively guided trial-and-error procedure in which the computer cooperated, turning up flaws in the reasoning or revealing unexpected turns in the solution.*
>
> (Licklider 2003, p.75)

This method for computer assistance fits in well with the way Cross describes "design thinking"; designers typically need to externalize their ideas:

> *The activity of sketching, drawing or modeling provides some of the circumstances by which a designer puts him- or herself into the design situation and engages with the exploration of both the problem and its solution.*
>
> (Cross 2011, p.12)

Systems that permit a collaborative conversation between a human and computer are often referred to as **mixed-initiative** (Allen et al. 1999; Novick & Sutton 1997). The first known use of this term is in reference to Jaime Carbonell's SCHOLAR system, a computer-aided instruction tool that allows the student and computer to engage in a "mixed-initiative dialogue" (Carbonell 1970). Such a dialogue is characterized by the ability of the human student to ask questions of the computer, just as the computer asks questions of the human.

Negroponte expands upon the idea of a computer assistant in the context of design in *Soft Architecture Machines*, describing a need for "computer-aided participatory design" in which a computer can act as an objective expert that can help a novice design his own house to his own needs, without pushing a particular agenda. Negroponte calls this roll of the computer that of a "design amplifier": a "surrogate *you* that can elaborate upon and contribute technical expertise to *your* design intentions" (Negroponte 2003, p.360). These amplifiers must act as an "intelligent extension" of the human. Gingold further explores this concept of a design amplifier, describing magic crayons that are easy to "pick up" and use to express different creative ideas, while also being sufficiently expressive and intelligent to make the user feel as though she has expressed herself exactly as she wished to, though she lacks the expertise to do so (Gingold 2003). *Spore Creature Creator* (Maxis 2008b) is a

system that embodies his idea of *magic crayons*, in which players can create complex, animated 3D models using a simple yet highly expressive tool.

Ivan Sutherland's *Sketchpad* system (Sutherland 2003) is perhaps the first example of an intelligent design tool. Sketchpad is a CAD tool that allows users to specify constraints on their designs, such as forcing lines to be parallel and intelligently resizing shapes, and to reuse their designs in other design projects. Sketchpad paved the way for other Computer-Aided Design (CAD) systems; for example, there are tools are available for circuit design and analysis (Nagel & Pederson 1973), robot behavior design (Vona 2009), kitchen design (Morch & Girgensohn 1991), and building 3D architectural models (Google 2000) among many others. Some of these tools are intended for novice designers, who look to the computer to provide guidance and domain expertise. Others are for professional designers, whose jobs are made much simpler by using a computer as a subservient assistant who can shoulder some of the design burden.

These tools are necessarily domain-specific: Gerhard Fischer calls such tools "domain-oriented design environments" (DODEs), noting that domain-specificity "reduces the large conceptual distance between problem-domain semantics and software artifacts" (Fischer 1994). Fischer describes DODEs as:

> *…cooperative problem-solving systems in which the computer system helps users design solutions themselves as opposed to having an expert system design solutions for them…*
>
> (Fischer et al. 1993)

Therefore, according to Fischer, it is vital for DODEs to incorporate a critiquing system, where the computer can act as an expert in the domain, capable of analyzing and reasoning about properties of the design. It is also possible for the computer to act as an "audience" for the designer, rather than a domain expert, in which the critic is actually an AI acting as the recipient of the product being designed. Riedl and O'Neill describe this scenario as a potential fifth role for computers in creativity support tools, extending Lubart's proposed four main roles described earlier (Riedl & O'Neill 2009).

The key concept in both Fischer's and Riedl and ONeill's work is that a design tool should be able to provide instant feedback to the designer. Whether this feedback comes in the form of critics who are domain experts or a simulated non-expert audience, it is most important that AI-assisted domain-oriented design environments be able to offer the designer a way to understand the worth of the artifact being created. This feedback can also be from the human designer herself, as in Bret Victor's game design tool that allows a programmer to instantly see the impact of adjustments made in the code without recompiling, supports pausing, "rewinding", and forward simulating time during play to examine crucial game moments, and provides GUI elements such as sliders for changing values such as movement speed and gravity (Victor 2012).

The work presented in this dissertation, specifically the creation of the Tanagra intelligent level design assistant, builds on all of the work described in this section. Tanagra acts as a colleague in the design process, using a mixed-initiative communication model in which the human and computer take turns working towards a common design goal. It allows a level designer to create content for games by controlling an expressive content generator which,

although it displaying biases towards creating certain kinds of content, will never promote its own ideas of what the level should look like above the ideas of the human designer. It supports intelligent design operations such as pacing changes, and can provide instant feedback to the designer in terms of the playability of the level. Tanagra is a step towards the future of AI-assisted design tools in the domain of game design.

CHAPTER 3

DESIGN SPACE ANALYSIS*

Procedural content generators that play a meaningful role in the game design process must have a semantic understanding of the domain they operate in. In the case of building PCG-driven tools for designers, the knowledge representation used by the generator forms the foundation for communication with the human designer. When PCG is used as a mechanic in game design, the ways in which a game designer can control the generator and use it to react to player's choices are greatly influenced by the way the generator understands its domain.

An analysis of the design space contains a set of all the different components the generator can use to construct content and information about how these components influence gameplay through the player's interactions. It also contains rules and constraints for how components can be combined, both in terms of physical limitations (e.g. a spring must always sit on top of a platform) and gameplay considerations (e.g. how to control the pacing of a level). This chapter provides an overview of existing work in performing genre-based analysis of games, and describes the representation of levels used in both the *Launchpad* and *Tanagra* generators.

* The rhythm-based framework for the analysis of 2D platforming games described in this chapter has been previously published (Smith et al. 2011b; Smith et al. 2008b).

1 Existing Models of 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 (Salen & Zimmerman 2004). Books that address level design primarily address its general principles. *Level Design for Games* (Co 2006) discusses the importance of spatial layout and atmosphere. *Level Design: Concept, Theory, and Practice* (Kremers 2009) describes two hierarchies of level design considerations; one describes designing for player goals, the other describes designing the structure of levels. Level design goals involve scenarios, gameplay moments, and gameplay actions. Gameplay actions are roughly equivalent to "beats" in our analysis of rhythm groups, but the analysis we provide does not directly address the relationship between player goals and player actions; some of the structural hierarchies Kremers discusses are reflected in the cells and portals analysis of platformer level design. *Fundamentals of Game Design* (Adams 2009) and *Beginning Game Level Design* (Feil & Scattergood 2005) also largely focus on general issues, giving a couple of paragraphs on level design for specific genres.

However, while they are useful, abstract discussions of level design concepts and short descriptions of genre-specific level design are not sufficient to provide a detailed understanding of the structural relationships of elements in a level for a particular genre. The level representation described here must be powerful enough to drive a PCG system that can be expressive enough to be used by designers; a highly detailed analysis of level design for the genre of 2D platformer games is necessary.

While there has not been a great deal of analysis for genre-specific level design, there are some notable exceptions. Nelson's article on level elements in *Breakout* is a good example of the kind of detail needed to fully model levels (Nelson 2007). 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. Dormans performs a deconstruction of The Forest Temple level in *The Legend of Zelda: Twilight Princess* (Nintendo EAD 2006b) to analyze both the mission and space structure; this allows him to represent meta-level, experiential concerns such as player goals independently from the construction of the levels themselves (Dormans 2012). This is done with a similar goal to our separation of rhythm from geometry in our own analysis of 2D platformer levels.

There is also work in identifying and analyzing design patterns in levels (see also Chapter 2, Section 1.1.1). Hullett & Whitehead identify a number of common idioms used in first person shooter (FPS) levels (Hullett & Whitehead 2010), such as arenas and sniper nests. While difficult to compare analyses of such different genres, these idioms seem analogous in scale to the cells in our model for platformer levels, as each idiom contains a number of different, potentially disjoint player actions. The design patterns used in Tanagra (see Chapter 5, Section 2.2) operate at a smaller scale than the FPS design patterns. Milam & El Nasr identify patterns in levels covering a number of different genres (Milam & El Nasr 2010); 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 articles analyzing level design aspects of platformers in particular. Boutros writes about common design goals in the best-selling platform games (Boutros 2006), focusing especially on visuals, controls, rewards, and challenges. He analyzes the first 5-10 minutes 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 (Dormans 2005). Missing in these two articles 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 (Byrne 2004) because it is the key to the player enjoying the level. Nicollet also discusses the role of challenge in dexterity-based games (Nicollet 2004). 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.

2 Deconstructing Platformers

The model used in this work for 2D platformer levels was constructed from an analysis of a number of games that are exemplary of the genre, including *Super Mario World* (Nintendo EAD 1990), the *Sonic the Hedgehog Series* (Sonic Team 1991; Sonic Team 1992), *Yoshi's*

Island DS (Artoon 2006), and *Donkey Kong Country 2: Diddy's Kong Quest* (Rareware 1995). 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 in this game 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, but many are hidden from the player and the challenge comes in discovering them through a thorough exploration of the physical space of the level.

Even within a single game, there are a variety of ways in which levels are composed of similar elements to produce different player experiences. For example, Figure 5 shows three levels from the same game: *Super Mario Bros.* (Nintendo Creative Department 1985). The

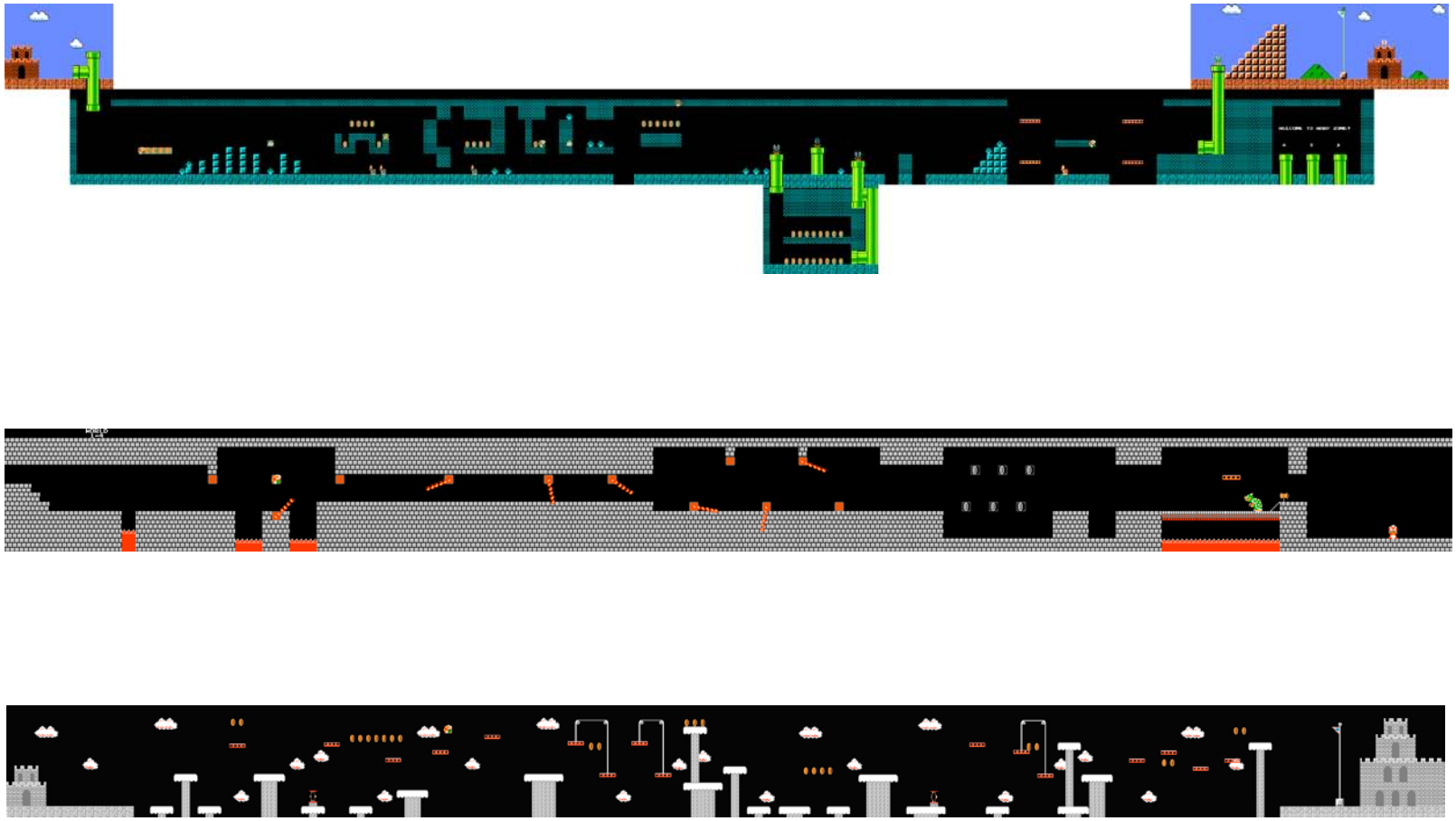

Figure 5. **Three Worlds from Super Mario Bros. showing different kinds of player behavior.** Top: World 1-2; Middle: World 3-4; Bottom: World: 6-3. World 1-2 involves the player mostly running and jumping; World 3-4 has the player take mostly *wait* actions; World 6-3 involves a combination of jumping and waiting. Maps are courtesy http://www.vgmaps.com

first level, World 1-2 involves the player constantly running and jumping over obstacles, searching for coins and either avoiding or killing enemies. The second level, World 3-4, has the player constantly stopping and starting, as she waits for the fire sticks to rotate into a position where it is safe to move under them. And the third level, World 6-3, involves a combination of jumping and waiting—there are platforms that counterweight each other, so when the player jumps onto one, she must wait for her platform to lower and the other to rise in order to jump onto the next one.

Despite the many differences between level styles, let alone game styles, there are a number of similarities in the games' component elements and how those components are 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.

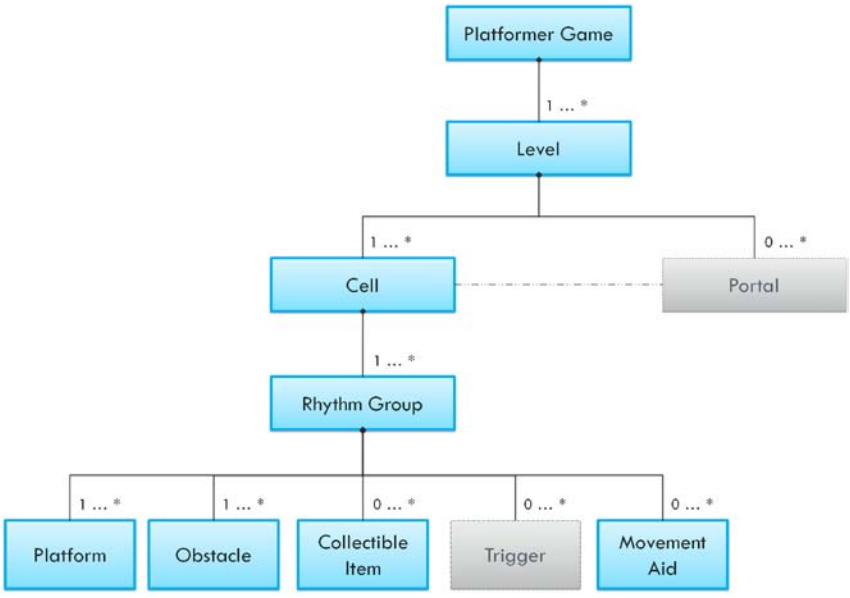

Figure 6. **A diagram of the rhythm-based level framework.** This conceptual model shows how the components in a level fit in to a more abstract struture. The thick-border boxes highlighted in blue are implemented in the systems described in this dissertation.

The remainder of this chapter consists of a detailed analysis of the levels in 2D platformer games. The diagram in Figure 6 describes the hierarchy of the kinds of level components found in the levels and their organization into the level's structure.

3 The Composition of Platformers

Platformer levels are composed of a variety of elements that the avatar interacts with during the course of play. This section describes the properties of these elements.

3.1 The Avatar

The avatar is the character that is controlled by the player, and has two major abilities in all platformer games: **run** and **jump**. Although there are sometimes multiple playable characters, such as Diddy and Dixie Kong in *Donkey Kong Country 2*, one player can usually control only one character at any given time. Avatar choices are sometimes merely cosmetic, but there can be 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; Baby Mario allows Yoshi to dash, and Baby Peach allows Yoshi to glide.

3.1.1 Physics Properties

The player tends to have control over the avatar's horizontal movement, and limited control over vertical movement through jumping and crouching. While an avatar's specific movement abilities vary per game, but can be approximated using a simple, ballistics-based physics model. This model includes the avatar's size, maximum movement speed, and initial jumping velocity.

3.2 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:

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.

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.

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.

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.* (Nintendo EAD 2006a). Daniel Boutros gives a detailed account of the design and structure of reward systems in a variety of platform games (Boutros 2006).

Collectible items have a great deal of impact on the player's satisfaction in playing the game (Nicollet 2004), 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 (Figure 7).

Triggers. Triggers are interactive objects or gameplay actions that change the state of the level. For example, Figure 8 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* (Armor Games 2008) 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.

Figure 7. **Example uses of collectible items.** Left: 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 four 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. Right: Collectible bananas literally point the way to a hidden area in *Donkey Kong Country 2*.

Figure 8. **Trigger use in *Super Mario World.*** 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.

4 The Rhythm-Based Structure of Platformer Levels

Section 3 describes the components that are common to all platformer levels, but in order to build a controllable level generator it is also important to understand the composition of levels, their internal structure, and the relationship to player experience. When level designers create levels from sets of components, they do so with a view to player enjoyment. Therefore, it is important to make sure that the structural portion of this

framework for analyzing and generating levels takes into account aspects of why platformers are engaging.

Rhythm and pacing are key to the player's enjoyment of a game (Bleszinski 2000) and also contribute significantly towards the difficulty of platformer games (Nicollet 2004). Therefore, the base structural unit in our level representation, the "rhythm group", directly represents the pacing of a level. Rhythm groups are short, non-overlapping sets of level components that encapsulate an area of challenge. These groups are inspired by Compton and Mateas's discussion of rhythm in platformer levels (Compton & Mateas 2006), who in turn were inspired by a method for representing rhythmic structure in music (Iyer et al. 1997). Rhythm groups help identify challenging areas of a level and understand what makes them difficult. Each rhythm group consists of a rhythm of player actions, and level geometry that corresponds to that rhythm.

The main building block of a rhythm is the **beat**. A beat in a rhythm corresponds to a single action the player takes during play. 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 player feels a rhythm while hitting the buttons on the controller. 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 staircase.

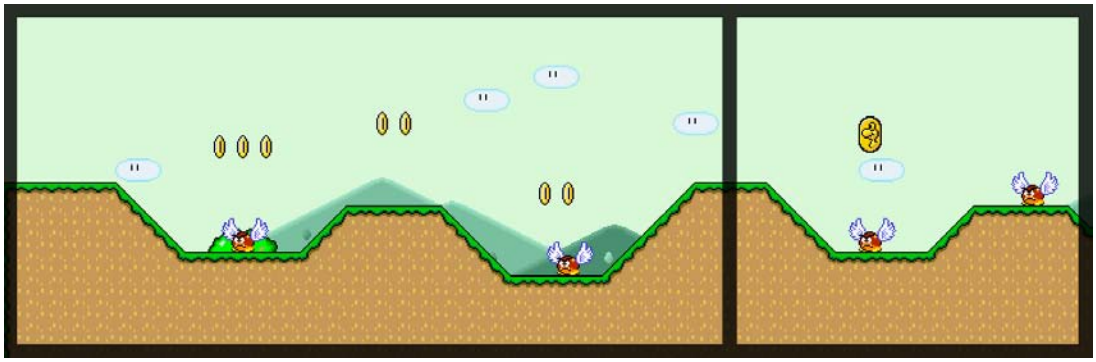

Figure 9. **Rhythm group example.** This segment of a *Super Mario World* level shows two rhythm groups, each surrounded by a black rectangle. The rhythm groups are separated by a platform where no action is required to be taken, allowing the player a brief rest before moving on to deal with the next obstacle.

Figure 9 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.

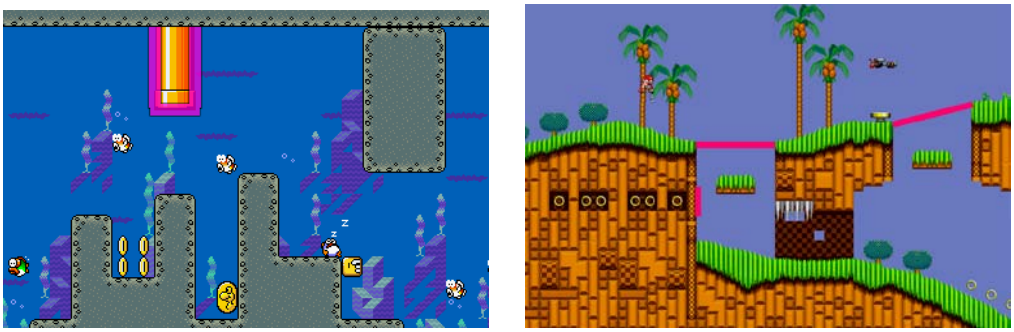

Figure 10. **Cells and Portals.** Two examples showing cells and portals. The upper figure shows a scene from *Super Mario* World with a single cell and a pipe that forms a portal out of it. The lower example is from *Sonic the Hedgehog 2* and shows three cells: the upper area, the lower area, and the 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.

4.1.1 Player Choice and Non-Linearity

Non-linearity in platformer levels is represented by cells and portals. This non-linearity 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 non-overlapping, 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. Figure 10 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 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.

5 Case Study: Sonic the Hedgehog

The rhythm-based structural model defined in this chapter provides a common vocabulary and framework for analyzing how existing levels are structured. This section uses our framework to describe a section of the Emerald Hill Zone, Act 1 level from *Sonic the Hedgehog 2*. Figure 11 is divided into labeled rhythm groups and portals. Each group is also lightly shaded, indicating that it forms part of a cell; cells are color-coded pink (groups 1 and 2), green (groups 3-5), and orange (groups 6-8).

1. This rhythm group is part of a cell that continues from the left, outside the range of this area. It contains a check point and several collectible items for the player to reach.
2. This rhythm group is part of the same pink cell as rhythm group 1, reachable by riding the moving platform from rhythm group 4. It contains a collectible item containing a power-up.
3. This rhythm group has the player run uphill and hit a spring to be able to reach the rings and checkpoint in the top left corner (thus entering a new cell, off to the left). The grassy areas that the avatar runs across are platforms in this section, both in the top left

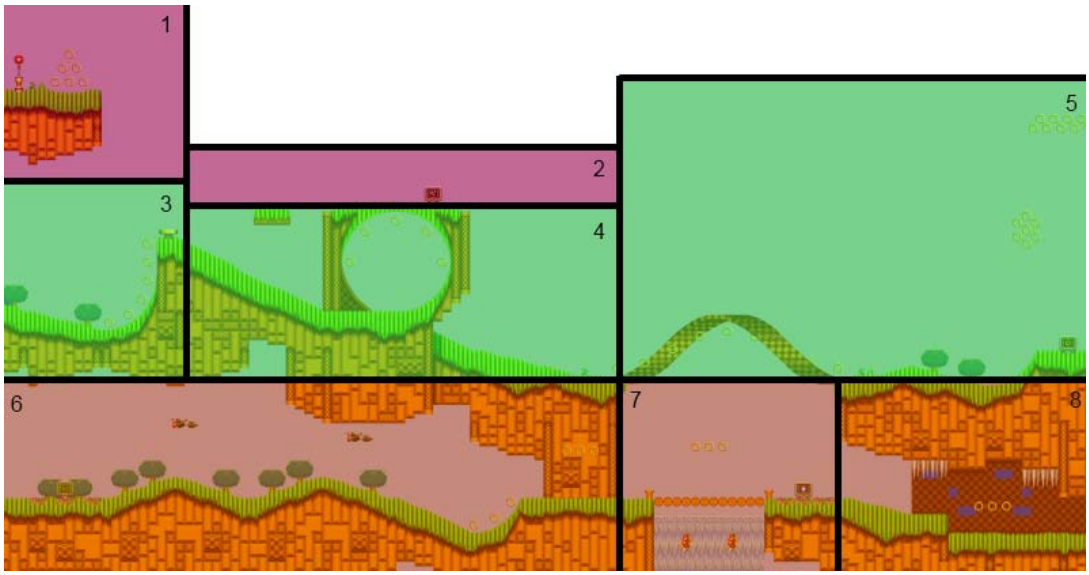

Figure 11. **Structural Analysis of Sonic the Hedgehog 2.** This is a segment of a level from *Sonic the Hedgehog 2*, split into rhythm groups marked by solid black lines and cells marked by background colors.

corner and the bottom region. There are no obstacles in this rhythm group. The spring at the end of this rhythm group is a movement aid. The coins are collectible items, and there are no triggers.

4. Another rhythm group in the same cell, this time giving the player several options. The platform on the left moves up and down, allowing the player to jump up to the top of the loop and collect a power-up (thus entering the pink cell). The player can then jump back down and complete the loop, rolling down towards the twister. If the player does not have enough speed to complete the twister at the exit of this rhythm group, he will enter the orange cell at rhythm group 7. Thus, there is a rhythm group boundary between groups 4 and 5.
5. This rhythm group begins with a "twister" platform. Even though it is possible to fall off the twister, no damage is done to the avatar. If the player has enough speed to

complete the twister, he will also have enough to collect the lower set of coins. The group ends with the player collecting a box containing ten rings.

6. This lower rhythm group is part of the orange cell. The pace in this group is slow relative to the rest of the level; the player has some speed but must only make one or two jumps to kill the enemies. The rhythm group ends for two reasons: firstly, there is a cell boundary that forces the end of the rhythm group. However, even without that cell boundary it would still end here because the rhythm changes from being one that primarily has the player running to one that has the player jumping more often.
7. A rhythm group characterized by timed jumps. The player must avoid the fish that jump up towards him from under the bridge; correctly avoiding the fish will reward the player with some rings. Failing to avoid the fish will reduce the player's ring tally to zero, but the coins above the fish serve as a guarantee of retaining at least some rings as a shield for the rest of the level.
8. This last rhythm group consists of spikes that are easy to avoid as long as the player does not accidentally jump into them.

Our analysis shows some interesting aspects of level design for *Sonic the Hedgehog*. The designers provide many choices for the player; these choices provide challenge since the main goal of the levels is to complete them quickly. The player must make the optimal choice to complete the level quickly.

Rhythm group size and allocation reflects the speed with which the player moves. Groups 3 and 4 are quite large, making up their own cells. The size of these groups is deceiving,

though; the player is required to jump no more than twice in each group to avoid obstacles, collect items, or destroy enemies.

We also see that the designer provides a second chance to players who fail to pick up enough speed to complete the twister. Region 5 shows the portal that lets the player transition to the lower right cell. Staying on the top path may be preferable, since there are a number of coins available to collect at the end of the top right cell. However, failing to complete the twister still provides compensation in the form of a temporary shield power-up. Indeed, the player may deliberately choose to fall through the twister and collect that power-up if he thinks it will help him later in the level.

6 Discussion and Future Work

The rhythm-based representation described in this chapter is well-suited for describing dexterity-based 2D platformer levels, and has been an effective foundation for building two different rhythm-based generators: Launchpad (described in Chapter 4) and Tanagra (described in Chapter 5).[*] To my knowledge, it is the first framework for describing, analyzing, and constructing 2D platformer levels at many layers of abstraction: from individual player actions to the larger-scale design concern of paths through a level. Other work in analyzing levels in this genre has generally focused at only one layer of abstractions,

[*] Both generators are beat-driven, geometry is generated based on a beat structure that is either generated (Launchpad) or specified by the designer (Tanagra). However, the precise definition of a beat in these two sytems is subtly different. In Launchpad, a beat corresponds to the exact moment in time that an optimal player will push a button to begin an action. Tanagra's definition of a beat is more relaxed, and instead refers to a period of time in which the player is guaranteed to take an action.

e.g. patterns of geometry combinations or identifying common types of challenges (Boutros 2006; Dahlskog & Togelius 2012).

Focusing on rhythm when analyzing levels allows us to understand how pacing changes along different paths through the level. In future work, it may be possible to automatically identify rhythm groups from data collected from players; this opens up many directions of new research in comparing level pacing in level progressions or even across different games. As with any descriptive framework, however, there are limitations to what our rhythm-based model can cover, and there is the potential to expand it to cover new aspects of level design. The remainder of this section discusses potential future directions for expanding the framework, which in turn would have an impact on the capabilities of both Tanagra and Launchpad.

There are some important subtleties to the physics and "feel" of platforming games. Swink provides a detailed analysis of the movement mechanics for the avatar in *Super Mario Bros*., including how there is a separate "falling" gravity from normal gravity, and a slow ramp-up to maximum movement speed, two aspects of movement that Swink claims make Mario more expressive than Donkey Kong (Swink 2008). Adam Saltsman provides a similar analysis of the physics, collision, and camera design decisions in *Canabalt* (Saltsman 2010). Such physics details are abstracted away in the ballistic physics model for avatars, and camera position and movement is not considered in the framework at all.

The separation of rhythm from geometry placement is a powerful abstraction, as it allows us to reason about timing and pacing information independently from particular level

components. However, there is no current way in the framework to represent design goals that motivate different pacing choices, or reasons why certain design choices are made. Levels are the primary mechanism by which a player progresses through a game, and learns new aspects of the game's design, but the representation described in this chapter provides no means for describing level progression or player skills. Cook's "skill atoms" provide a method for representing mechanisms for the player learning different aspects of a game's design through actions (Cook 2007); integrating this sort of framework for player learning with a structural framework for level design would improve the framework's expressiveness with regard to player experience.

CHAPTER 4

LAUNCHPAD: DESIGNER-INFLUENCED LEVEL GENERATION*

This chapter describes Launchpad, a level generator for 2D platformer games that uses the concepts of rhythm and pacing explored in Chapter 3. Launchpad is the level generator underlying *Endless Web*, described in Chapter 6.

1 INTRODUCTION

The goal in creating Launchpad was to design a level generator that provides a user with meaningful, global control over the levels it designs. The majority of level generators prior to the creation of Launchpad did not provide explicit control over output that was accessible to non-technical designers; instead, altering the output of the generator is accomplished through editing the generation code and tweaking parameters that do not necessarily have a clear relationship to the output of the generator. A user can control Launchpad by providing parameters guiding both the rhythms that the level should be composed of and the frequency of different geometry components across the level, as well as a general path that the level should follow through space. Communication between the user and generator is one-way; there is no ability to change a level after it has been created. The output from Launchpad is a set of playable levels, sorted by how well they fit the user's stated requirements. Launchpad can also be used to create a single rhythm group that fits well to

* Portions of this chapter have been previously published in a Transactions on Computational Intelligence and AI in Games (TCIAIG) article (Smith et al. 2011b; Smith et al. 2009c).

the parameters provided to the system; this is how Launchpad is used in *Endless Web* (Chapter 6).

Launchpad minimizes the amount of content that must be manually authored by working from small level components equivalent to what a designer would be manipulating in their level editor, rather than fitting together large chunks of manually authored content. These elements are all drawn from the analysis of the composition of platformer levels described in Chapter 3, e.g., platforms, spikes, enemies, and springs. Level elements are tagged with the action the player performs to use them, and are combined to fit to a generate rhythm. Launchpad is capable of creating a wide variety of levels, even for the same sets of input parameters.

Launchpad's expressive power comes largely from its separation of the pacing structure of levels from their geometric composition. This is similar to the basis of Dormans's later work in procedural level generation for action-adventure games; he also takes a two-layer, grammar-based approach to level generation, first producing missions using a grammar, then using shape grammars to create playable levels from generated missions (Dormans 2010). 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 non-geometric structure based on intended player actions. This idea is well expressed by Ashmore and Nitsche 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" (Ashmore & Nitsche 2007).

This chapter describes Launchpad's algorithm details, provides examples of the levels that can be created using Launchpad, and discusses the design tradeoffs that come from using a generate-and-test approach to content generation and potential extensibility for the generator. An evaluation of Launchpad's controllability and expressiveness appears in Chapter 7, an evaluation of the manipulation of Launchpad's generative space as a mechanic in the PCG-based game *Endless Web* appears in Chapter 6.

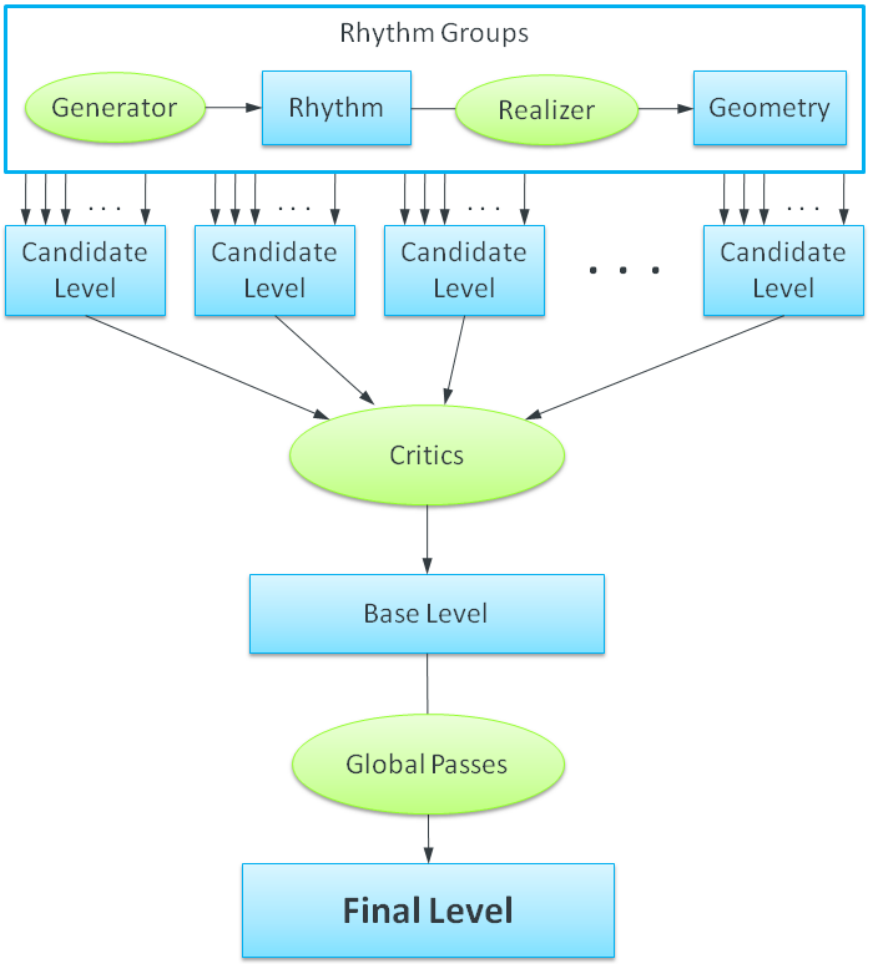

Figure 12. **Launchpad architecture diagram.** Blue boxes denote artifacts produced by the system; green ovals represent components of the generator, each of which is influenced by design parameters.

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 distribution for different actions types assigned to beats.
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.

Table 1. **Launchpad parameters.** The parameters for Launchpad that a user can manipulate.

2 GENERATING LEVELS

The system architecture for Launchpad is shown in Figure 12. Individual rhythm groups are generated using two grammars: one for rhythm generation, and another that consumes a rhythm to produce geometry. These rhythm groups are grouped together to form a pool of candidate levels, which are then analyzed by a set of critics to find the level that best matches the designer’s specified parameters. Finally, the levels that pass the critics undergo a final set of global decoration passes to produce the final level.

2.1 Rhythm Group Generation

Rhythm groups are generated using a two-stage, grammar-based approach. The first stage creates a set of beats, where each beat corresponds to a particular player action. The second uses a grammar to realize this set of actions into corresponding geometry according to a set of physical constraints. This creates a single rhythm group; many unique rhythm groups are used in creating levels.

2.1.1 Rhythm Generation

Rhythms have three main properties that can be controlled by the designer: 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 1), then generates an appropriate beat timeline by using the parameters to determine how many beats should be in the rhythm and how far apart they should be spaced. Example timelines for different combinations of these properties are shown in Figure 13. Each beat of the rhythm marks a moment when the player takes an action by pressing a key or button on the controller.

Beats are then assigned action types, based on the *action probabilities* passed in to the

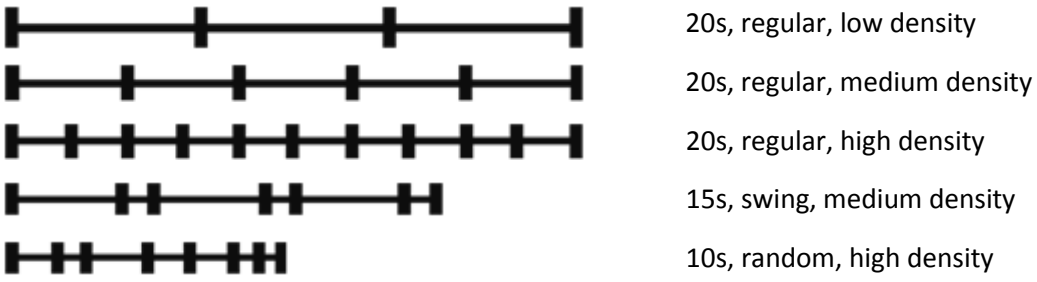

Figure 13. **Example rhythms**. These timelines show the effects of varying the length, type, and density of a rhythm. Lines indicate the length of the rhythm, and hatch marks indicate the times at which an action will begin.

generator. Players can take two types of action: “move” and “jump”; these are the two core movement mechanics identified in 2D platforming games. These actions also have an ending time, corresponding to when the player lets go of the button. This process produces rhythms such as the following, where the numbers after actions correspond to the start and end times of the action. A graphical representation of this rhythm is shown in Figure 14.

```
move  0     5
jump  2     2.25
jump  4     4.25
move  6     10
jump  6     6.5
jump  8     8.75
```

In this example, the player starts moving at 0 seconds, and continues moving until 5 seconds have elapsed. While moving, the player jumps once at the 2 second mark and again at the 4 second mark, with each jump key press lasting 0.25 seconds. The player then pauses moving at 5 seconds and begins moving again at 6 seconds, until the rhythm ends. While moving the second time, the player jumps once at the 6 second mark and again at the 8 second mark,

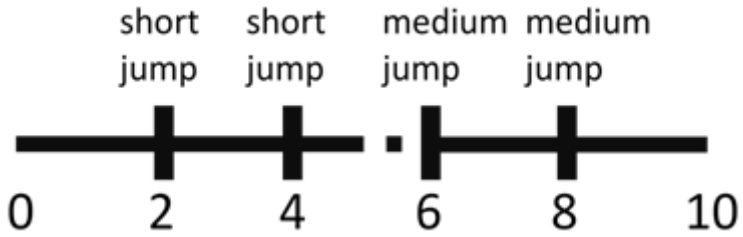

Figure 14. **Rhythm with actions.** This timeline corresponds to the example rhythm specified above. The solid line denotes time that the avatar is moving, the dashed line is time that the avatar is not moving.

this time holding down the button for 0.5 seconds and 0.75 seconds respectively, both longer times than the button was previously held.

Since different button hold times influence the height of the avatar's jump, it is important to record the amount of time the button is held. For example, the player may hold the button down for only 0.25 seconds, resulting in a very short hop, or may hold the button down up to 0.75 seconds for a much longer jump.

2.1.2 Physics Constraints

Once Launchpad has generated a rhythm, it then realizes this rhythm into level geometry using a grammar and a simple model of game physics. The physics system is parameterized; a designer can specify the size of the avatar and all level components, the additional velocity imparted by springs and jumping on top of enemies, and the avatar's maximum movement speed and initial jump velocity. These aspects of the physics system are specified independently from the rest of the generator, allowing a designer to easily experiment with different values for the physics system and still receive playable levels. Altering the physics parameters as a game mechanic was explored in the *Rathenn* prototype for *Endless Web*, described in Chapter 6, Section 5.1.2.

The physics system uses the parameters provided by the designer to calculate the height the avatar can jump given a short, medium, or long jump button press; the model assumes that the initial jump velocity will be applied for as long as the jump button is held. For jumps, the model includes the in-air time for each jump type, the 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—a common physics feature for *Mario*-style platformers (Swink 2008). More advanced player physics, such as double-jumping or wall-jumping, are not supported.

All of this information is used to process rhythms created by the rhythm generator into input suitable for the geometry grammar. This processing guarantees that all levels will be playable; i.e., there will be no impassable obstacles. 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 rhythm given above forms the following set of states and jump queue:

```
Movement States:  [5, moving], [1, waiting], [4, moving]
Jump Queue: [2, short], [4, short], [6, medium], [8, long]
```

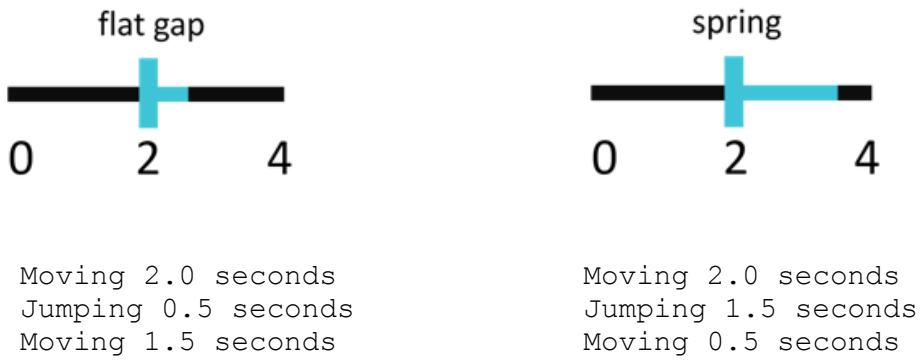

Figure 15. **Creating movement states.** Two different types of jump can contribute to different movement and jump state lengths. The blue area is the amount of time consumed by the jump being in the air.

However, the jump length merely tells how long the player has held the jump button. The physics system uses the jump length to determine how long the avatar will be in-air for each potential jump type. A jump across a flat gap takes considerably less air-time 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), that a jump across a flat gap takes 0.5 seconds, and that a jump onto a spring takes 1.5 seconds, processing the first jump in our example could result in one of the two configurations shown in Figure 15. However, if the second jump were to occur at 3 seconds rather than 4 seconds, the jump onto a spring would no longer be a valid jump, as it wouldn't 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.

```
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)
```

Figure 16. **Geometry generation grammar.** Player states derived from the generated rhythms are the main non-terminals in this grammar, bolded here. Other, intermediate non-terminals (italicized) are used for expressing further variety in different types of geometry components.

```
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
```

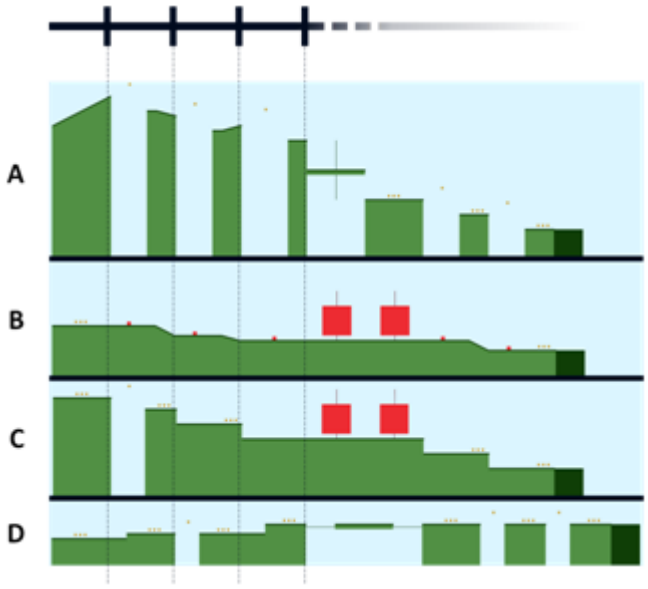

Figure 17. **Geometry interpretations of a rhythm.** This figure shows four different geometric interpretations of the provided rhythm. Small red boxes denote enemies to kill, large red boxes are stompers that follow the associated line, and platforms on green lines are moving platforms that follow that path. The large, dark green platform at the end of the rhythm group is the joiner from this rhythm group to the next.

2.1.3 Geometry Realization

The movement states that are not fully consumed by jumping and the queued jumps form the non-terminals in the geometry generation grammar (see Figure 16). 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 designer frequency of components as specified by the designer (Table 1).

Figure 17 shows an example of a rhythm and four different geometries that can be generated from it. Parts (a) and (d) of this figure show how moving platforms consume a wait-move-wait; other interpretations of two wait-moves in a row are shown in parts (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 that occur after waits no longer "line up" with the sample rhythm. This figure also shows many different geometric interpretations for each jump action.

3 CRITICS

Complete levels are generated by piecing together rhythm groups by connecting them via a joining platform. This gives the player an opportunity to rest before beginning the next challenge, which is an important aspect of level pacing (Nicollet 2004). Rhythm groups can optionally be repeated before this rest area, according to a probability set by the human designer (Table 1), which can provide additional challenge and more visual consistency. Each level is the length of the control line specified by the user, as described below.

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 over-generation; 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.

Launchpad uses these critics by over-generating levels using the rhythms and grammars. First, 100 rhythms are generated according to the rhythm and action probabilities set by the designer (see Table 1). Next, Launchpad creates 1000 "candidate" levels[*] by creating rhythm groups from a randomly selected rhythm and piecing those rhythm groups together into full levels. These candidates are scored by the two critics described in this section; the designer can then choose a level based on its assigned critic score.

3.1 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* (Nintendo EAD 1990) and *Sonic the Hedgehog* (Sonic Team 1991) 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 Figure 18.

[*] The number of candidate levels that can be generated can be specified by the user of the system; we have found that 1000 candidate levels is an adequate number to get sufficient variety to satisfy the critics, while also being rapid enough to generate that the system responds quickly to designer demands. When generating rhythm groups for *Endless Web* (Chapter 6), the same number of rhythms are generated, and the critics are applied to 250 candidate rhythm groups, rather than entire levels.

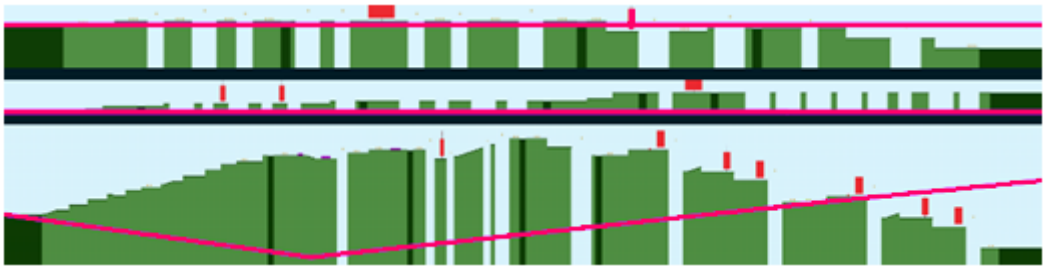

Figure 18. **Line critic scores.** Examples of levels that are different distances from a specified control line (shown in pink). The top level has a distance measure of 1.75, the middle a distance measure of 5.08, and the bottom a distance measure of 23.06.

3.2 Component Frequency Critic

The component parameters described in Table 1 are used in geometry generation as a weight for how often each component should be chosen by the grammar; 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 generated rhythm groups. Therefore, there is no guarantee that the observed frequency of components will match the desired frequency, just as in a series of 10 coin flips there is no guarantee that 5 will come up heads and 5 will come up tails, even though one would expect a 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 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 specified by the designer.

3.3 Combining Critics

There are many cases in which these two critics may 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. This is an additional parameter the designer can provide Launchpad before it creates any levels.

4 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 as a whole. Launchpad has two "global pass" algorithms that cannot run until the level has been placed together: 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 visual consistency for the 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 a decoration on a level, to guide a player across jumps and to fill long flat runs, similar to coin use in *Super Mario World* (see Chapter 3, Section 3.2). Two rules determine collectible item placement:

1. Place a group of coins along a long platform (of predetermined length) that has no other action than movement associated with it, and

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 probability for these coins being placed, and the number of coins that should be placed in each situation, are specified by the input parameters.

It would also be relatively 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. This comes from 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 so that the generator can react to their placement, rather than as decoration after the fact.

5 Discussion and Future Work

Launchpad is the first level generator to provide parameterized control over both compositional and experiential aspects of level design. Designers can manipulate parameters related to the physics of the world, the frequency of level components, and the desired pacing for the level. The parameterized physics model ensures that any generated level will be fully playable.

There are a few different ways in which Launchpad has been used in a final product. A demonstration of Launchpad as a design tool for complete levels is shown in Figure 19[*]. The

[*]A version of this demo modified to only include pre-generated levels is available in the supplemental materials and online: http://sokath.com/dissertation_supplement/launchpad/demo/platformer/

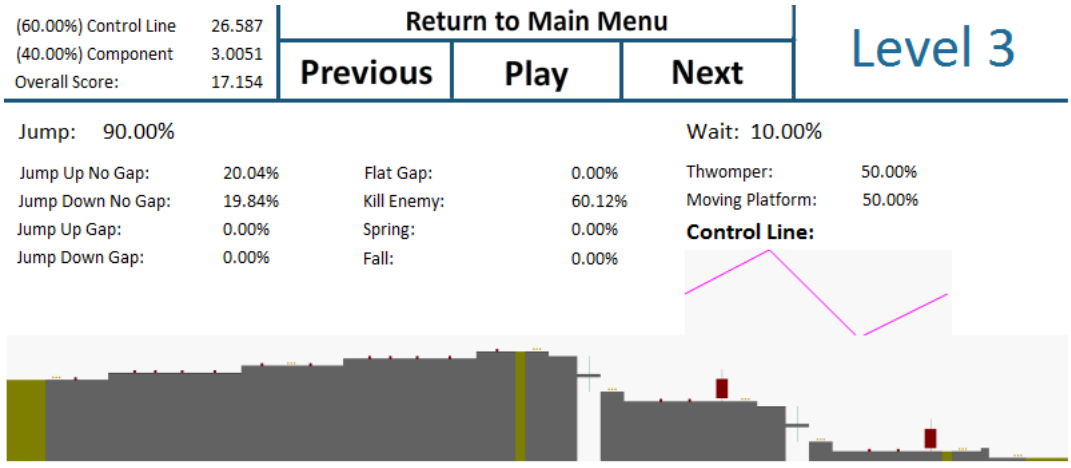

Figure 19. **A screenshot from Launchpad's online demonstration.** This is one of 10 levels that fits the parameters and control line specified by a designer.

demo includes 15 sets of pre-generated 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. A modified version of Launchpad is the underlying generator in the PCG-based game *Endless Web,* which is described in Chapter 6.

Launchpad has also been used as the level generator underlying Polymorph (Jennings-Teats et al. 2010), to which I contributed. Polymorph uses procedural level generation for dynamic difficulty adjustment. The system works by taking data gathered from players ranking the difficulty of generated level segments and learning which features of levels correspond to difficulty. These features emerge from combinations of components chosen by the generator; for example, a gap followed by another gap, or a gap followed by an enemy. Polymorph uses this model of difficulty to generate candidate level segments and rate their difficulty, then chooses the one of the most appropriate difficulty to the player's current

performance to appear “just in time” in front of the player as she moves through the level. Launchpad’s role in Polymorph provides an example of how flexible the system can be. Launchpad uses a universal model of player behavior (rhythmic actions) when generating levels, rather than focusing on generating a level for a particular kind of player. By taking a more general approach to level generation, Launchpad can be used in a variety of applications. One can imagine augmenting Launchpad with additional information, such as models of player behavior or data about difficulty, for either creating a personalized experience or showing additional information to a designer using the system.

Overall, the rhythm-based approach for level generation has been quite successful. Rhythm and pacing are important aspects of platformer levels, though 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.

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.

Launchpad is the precursor to the Tanagra level design assistant, described in Chapter 5. While Launchpad provides a sufficient amount of control for use in a game such as *Endless Web* or Polymorph, it does not provide nearly direct enough control for use at design-time. In Tanagra, we wanted to preserve from Launchpad the level of granularity at which the generator operates and the simplicity and emergent expressivity of grammars. Tanagra uses a similar rhythm-based and grammar-based approach to level generation, but with more reactivity—a level can be modified in real-time to accommodate changes made to pacing and geometry placement.

5.1 Future Work

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. Other components, such as triggers that impact 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* (Armor Games 2008), 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 (Dormans 2010) in creating missions and levels for *Zelda*-style adventure games offers encouraging results in grammar-based level generation for a different genre. However, I 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 (Cook 2007).

CHAPTER 5

TANAGRA: MIXED-INITIATIVE LEVEL GENERATION*

Creating a good level is a time-consuming and highly iterative process: the level may start as a simple sketch of the space, which is then filled in with specific geometry. Designers will typically play the level themselves many times before showing it to anyone else, checking that it is playable, engaging, and meets their expectations (Novak & Castillo 2008). Making a change to a small section of a level, such as moving a single critical platform, can have a significant impact on the design and require much of the rest of the level to be modified as well. The tools that are available to designers to perform these tasks, especially for 2D platformer games, do little to alleviate these problems. Most 2D level design tools are little more than digital graph paper—level designers place tiles for platforms, environmental obstacles, and NPCs into a simple grid. There is very little automation in these tools; any changes made to the environment must be done entirely by hand. Furthermore, the designer begins from a completely blank slate; there is no obvious way to begin designing a level. It is very difficult to rapidly prototype different design ideas using these tools.

This chapter presents Tanagra, a mixed-initiative level design tool operating in the domain of side-scrolling 2D platformer levels that incorporates procedural level generation to ease the design burden. Designers can request generated levels to start from, rather than starting from a blank slate. Whenever level geometry is moved, the rest of the level morphs to adapt

* Portions of this chapter have been previously published in a Transactions on Computational Intelligence and AI in Games (TCIAIG) journal article describing Tanagra (Smith et al. 2011f).

to changes made by the designer. The goal with Tanagra was to create the technology that can enable an accessible design environment that makes it easy for designers to brainstorm and prototype different ideas while being safe in the knowledge that all levels created in the tool will be playable.

The mixed-initiative approach to design, where content is created through iterative cycles between the human designer and a procedural content generator, capitalizes on the strengths of both human and computer designers. Tanagra's underlying level generator is capable of producing many different variations on a level more rapidly than human designers, whose strengths instead lie in creativity and the ability to judge the quality of the generated content. The generator's ability to guarantee that all the levels it creates are playable refocuses early playtesting effort from checking that all sections of the level are reachable to exploring how to create fun levels. Also, our gameplay-centric approach to level representation and content generation opens up possibilities for novel editing operations: in addition to controlling physical properties of the world such as platform placement, the designer is also able to control properties related to potential player behavior by influencing the pacing of the level. In Tanagra, the human designer and procedural generator work together in a collaborative way, each taking turns to build on the work of the other. The human and computer communicate via the rhythm-based representation for levels described in Chapter 4.

From Launchpad (Chapter 4) we learned that grammars and rhythm are good methods for creating a content generator for 2D platformers. However, the grammar used in Launchpad is not nearly reactive enough for use in a design tool. Designer changes to a level in Tanagra

can result in a ripple effect—in order to support a platform being moved to a new position, for example, much of the rest of the level might need to be changed. Or, the platform might be able to move slightly from the level's current configuration without much need for changing the rest of the level. Tanagra must also be able to decide to reuse geometry that the designer has already drawn into the level when appropriate. Thus, Tanagra requires a new kind of grammar that can be responsive to designer modifications while still being sufficiently expressive and able to create levels that are guaranteed to be playable.

To accomplish this, we used a combination of reactive planning and constraint solving to create a **reactive grammar.** We use A Behavior Language (ABL)[*] (Mateas & Stern 2002) for reactive planning, and Choco (Choco Team 2008) for numerical constraint solving. The use of reactive planning allows for authoring behaviors that correspond to grammar rules and that allow procedural generation to be interleaved with a human designer's actions. These behaviors monitor multiple aspects of the generator in parallel, and their hierarchical nature allows for more complex geometry patterns to be built up from simpler components. The geometric relationship between level components that are placed by the generator is expressed as a set of numerical constraints that must be satisfied, thus ensuring that the design tool will never allow for the creation of an unplayable level.

1 Related Systems

Author-guided level generation tends to place all authorial control over the generator at the beginning of the process (Hullett & Mateas 2009; Smith et al. 2011b; Tutenel et al. 2009),

[*] The version of ABL used in Tanagra was modified from the original version to support inheritance for Working Memory Elements, which allows the agent to reason about certain collective properties of geometry and constraints without needing specialized behaviors for each type of them.

occasionally allowing editing after the level is complete (Firaxis Games 2005). These systems are typically examples of parameterized generators (see Chapter 2). For example, the world builder for *Civilization IV* allows the scenario designer to set certain terrain parameters ahead of time, such as the size of the land masses, distribution of water and land, and climate. After the generator creates the initial world, the designer can modify the terrain according to her own desires. However, there is no way to request another map that respects the changes that the designer has made, or that only a part of the level be regenerated. The mixed-initiative nature of Tanagra means that the designer and computer can collaborate throughout the design process. An important exception is the SketchaWorld project (Smelik et al. 2010), which provides a mixed-initiative authoring environment for virtual worlds, including terrain editing and city building. This project faces many of the same design concerns as Tanagra in determining how best to have designers interact with a PCG system. SketchaWorld focuses on building large-scale virtual worlds which are created with no concern about player behavior within the world, whereas Tanagra focuses on building levels that dictate the core gameplay. To my knowledge, Tanagra is the first mixed-initiative design tool to address the interactive elements of a game rather than purely its fictional world.

Other AI-supported design tools include BIPED, which allows designers to rapidly specify prototypes of their games and view play traces from both human and computer players (Smith et al. 2009a). The system's strength lies in allowing a designer to view situations in which their game design "fails" by asking the system to show a playthrough that produces an undesirable result. Tanagra shares the goal of allowing designers to rapidly prototype

different level design ideas, but does not support any qualitative evaluation of levels, beyond guaranteeing level playability. QuestBrowser is a brainstorming tool for quest designers, intended to show different potential solutions that a player might think of to better inform the design (Sullivan et al. 2009b). Brainstorming is an important aspect of creativity support (Sternberg 1999), one which Tanagra fulfills by providing the ability to rapidly regenerate levels.

A mixed-initiative approach to level generation requires a new set of techniques, especially given the real-time nature of a design tool. Heavily search-based approaches, such as evolutionary algorithms, have been successful in offline level generation and adaptive content creation between play sessions (Shaker et al. 2010; Togelius et al. 2007; Togelius et al. 2010), but their reliance on an unbounded search process makes them too slow for a realtime tool. An interactive design tool requires the same, if not greater, amount of responsiveness as online PCG techniques. Grammar and rule-based approaches show a great deal of promise in this area (Müller et al. 2006); however, existing grammar-based design tools are confined to creating large structures that the player does not interact with. For example, *UnrealFacade* (Epic Games 2010) enables procedural design of buildings, but aspects of the design that are heavily tied to core gameplay are still entirely authored by hand (Golding 2010). Tanagra's division of the generation process into reactive planning and constraint solving permits the system to respond quickly to designer changes at the pattern- and beat-levels while still guaranteeing level playability.

Figure 20. **Tanagra level components.** These are the tiles used by Tanagra. Each level component is made up of one or more of these tiles.

2 Level Representation

The level representation used in Tanagra is based on that described in Chapter 3. Levels are represented in Tanagra as a series of beats, where each beat corresponds to a single player action. These beats contain level components, also referred to as *geometry*. Supported level components are platforms, gaps, springs, enemies, and stompers. In turn, each of these components is made up of a set of one or more tiles (Figure 20). Since tile maps are a common representation for 2D levels; all geometry that is drawn into the level is done at the tile representation layer, which provides a simple interface for the human designer to draw in level geometry.

Each type of level component has a number of constraint variables associated with it. A platform's variables are a start and end position and a width. Enemies, stompers, and springs have an x and y coordinate. Gaps have a width and height. Each component also has a length variable associated with it, which states the amount of time the player takes to interact with that component in an ideal playthrough (i.e., a situation in which the player character takes minimal time to traverse the level). This allows a single numerical constraint—that the sum of the lengths of all components equals the length of the beat they inhabit—to maintain the desired length and pacing of the level.

Similar to Launchpad, a physics model defines the maximum running speed, maximum jump height, and physics properties of level components, thus guaranteeing that all geometry placed into the level is playable and meets beat duration constraints. This physics model can be changed externally from the tool (before compilation); future work involves integrating the ability to make changes to the physics model in the tool to see level changes in real-time.

2.1 BEATS

As the building blocks of rhythms, beats are the underlying structure for Tanagra's level generator. A designer can control the pacing of the level by changing their length, adding, and removing them. A beat represents a single action that is taken by the player, such as jumping or waiting. Its primary role in the level design is to constrain the length of the geometry within it to the distance that can be traversed by the player in the duration of the beat, as calculated from the physics model. Beats are also a convenient way of subdividing the space for the generation algorithm, since geometry for each beat can be generated largely independently.

The action that the player takes can occur at any time between the start and end times of the beat. Each beat has the following properties: constraint variables for the start time, end time, and length of the beat (measured in milliseconds), and knowledge of its preceding and following beats. Beats also keep track of their entry and exit platforms; level playability is guaranteed through beat constraints that match up the exit platform of one beat to the entry platform of its next beat. At any time, the designer can add, remove, or modify a beat, which propagates any changes down to the geometry contained within it.

```
Beat(3)     → Staircase | Valley | Mesa
Beat(2)     → Gap(1) Enemy(2)
Beat        → Gap | Enemy | Spring | Stomper

Staircase   → Gap(1) Gap(2) Gap(3)
Valley      → Gap(1) Enemy(2) Gap(3)
Mesa        → Gap(1) Enemy(2) Gap(3)

Gap         → platform gap platform
Enemy       → platform enemy
Spring      → platform spring gap platform
Stomper     → platform stomper
```

Figure 21. **A grammar-like representation of Tanagra geometry patterns**. Numbers in parentheses correspond to the number of beats required (on the left hand side) or the number of the beat to fill with a pattern (on the right hand side).

2.2 GEOMETRY PATTERN LIBRARY

Level components are built up into patterns based on the action the player should perform during the associated beat. Taking advantage of ABL's HTN-like hierarchical structure, each pattern can build on another. These patterns can be considered equivalent to Launchpad's grammar production rules, where beats form the main non-terminals; see Figure 21 for their formal specification. However, complicating the grammar somewhat is the introduction of constraint solving—the terminals in this grammar are not instantiated objects in the level, but rather a set of variables for Choco to solve for to determine precise positioning. Furthermore, the multi-beat patterns have additional constraints across the beats. Patterns are implemented in Tanagra as a combination of ABL behaviors and Choco constraints, described in further detail in Sections 4 and 5. The general use of ABL and Choco for reactive grammars is discussed in Section 10.

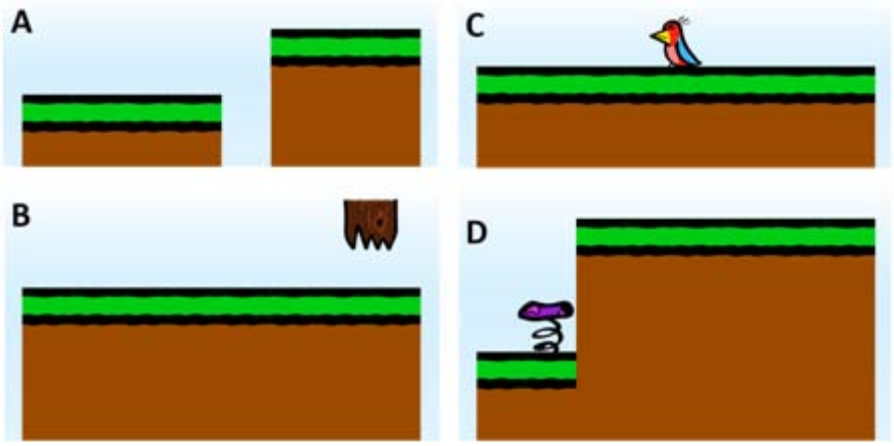

Figure 22. **Example instantiations of the four single-beat geometry patterns used in Tanagra.** (A) A gap between two platforms, (B) a stomper, (C) an enemy, and (D) a spring to a different platform. Patterns are described and produced using ABL behaviors. Note that there are many different configurations of each pattern; the precise placement of geometry is determined by the constraint solver (Choco).

At the base of the hierarchy are the following single beat patterns:

- Jumping over a gap from one platform to another
- Jumping to kill an enemy
- Jumping onto a spring
- Waiting before running underneath a stomper

Each of these patterns contains a single user action, and therefore spans a single beat. Gaps can be of variable width, from zero to the maximum length that the player can jump, and variable height, from the maximum height that the player can jump to its opposite value. Examples of each of these patterns are shown in Figure 22. Enemies, springs, and stompers each occupy only one tile, but can have different positions along the platform.

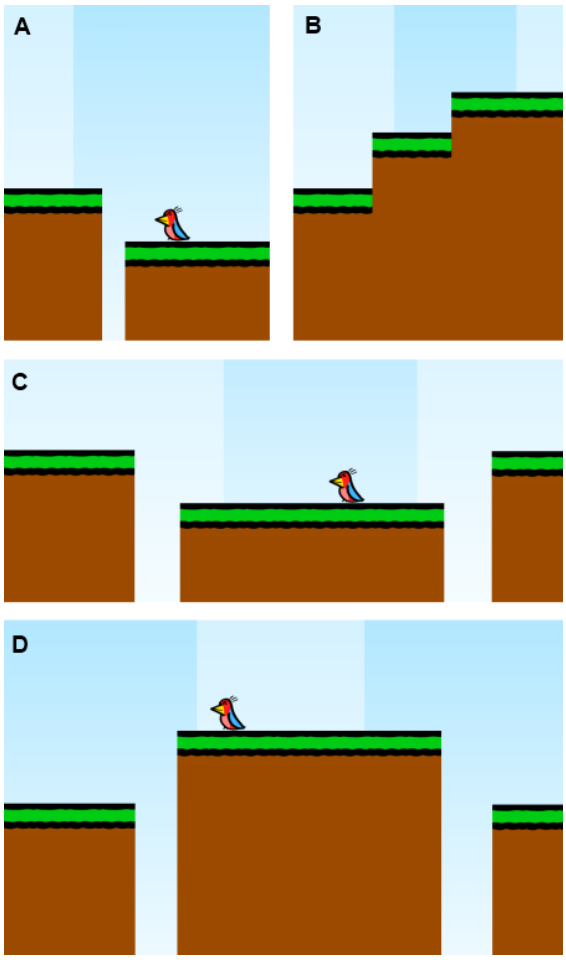

Figure 23. **Example instantiations of the four multi-beat geometry patterns used in Tanagra.** (A) A gap followed by an enemy, (B) a staircase, (C) a valley,and (D) a mesa.

Multi-beat patterns provide further structure to levels and mimic patterns commonly found in 2D platformers. They are composed of the single-beat patterns mentioned above with some additional constraints. The multi-beat patterns implemented in Tanagra, and the number of beats they span (in parentheses), are:

- Gap followed immediately by an enemy (2)
- A staircase, consisting of three gaps in a row, each of them the same width and height (3)

- A valley, consisting of a jump down, an enemy to kill, and then a jump back up (3)
- A mesa, consisting of a jump up, an enemy to kill, and then a jump back down (3)

Example instantiations of these patterns are shown in Figure 23. These more abstract patterns are straightforward to specify due to ABL's hierarchical nature and the separate specification of geometry placement and physics constraints. That all of these patterns span consecutive beats, however, is not a general requirement. For example, it would be possible in future work to specify a more abstract staircase pattern for an entire level, where every other beat contains a jump up over a gap, but the intermediate beats contain randomly selected geometry.

2.2.1 Pattern Constraints

As mentioned earlier, the non-terminals in our reactive grammar—the level components themselves—are not instantiated world objects but rather a set of constraint variables that are solved for by the constraint solver. This means that there is a great deal of variation within patterns in terms of configurations of geometry; a gap pattern has a gap with variable width and height. The variables for level components are subject to many kinds of constraints, further discussed in Section 4.2, including constraints that are dictated by the patterns in which their components are used. For example, consider a scenario in which the generator has decided to place a "staircase" pattern across three beats: b1, b2, and b3. The staircase pattern dictates that each of the gaps in b1, b2, and b3 have the same width and the same height. Furthermore, the gap pattern itself includes constraints on how the gap's width and height influence the positioning of surrounding platforms. Figure X illustrates the constraints present in a staircase pattern, assuming the following notation. There are three

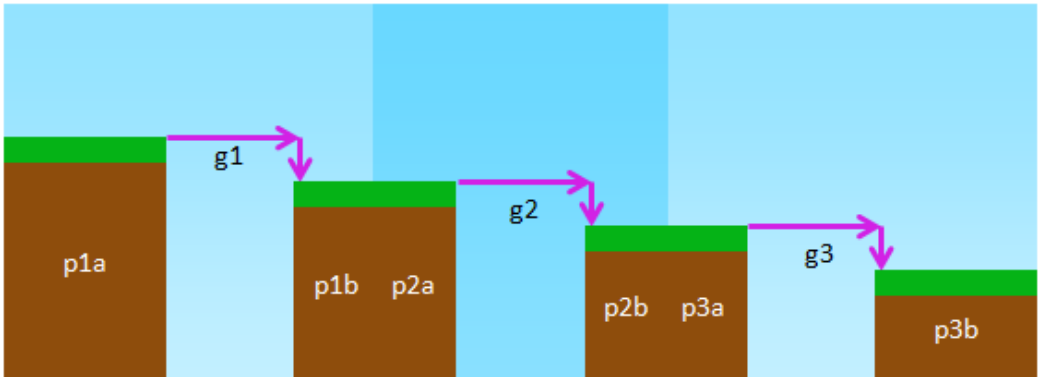

Figure 24. **Illustration of constraints placed by a Staircase pattern on gaps and platforms.** The three gaps are constrained to have the same width and height as each other, and each gap's width and height constraints the relative positions of their surrounding platforms within a beat.

gaps, one per beat: *g1*, *g2*, and *g3*. Each gap has two variables to consider: *width* and *height.* There are six platforms, two per beat: *p1a*, *p1b*, *p2a*, *p2b*, *p3a*, and *p3b*. Each platform has four variables to consider: *startX, startY, endX, endY.*

The Staircase pattern places the following constraints:

g1.width = g2.width = g3.width g1.height = g2.height = g3.height

Each beat's gap pattern places the following additional constraints:

p1a.endX + g1.width = p1b.startX p1a.endY + g1.height = p1b.startY

p2a.endX + g2.width = p2b.startX p2a.endY + g2.height = p2b.startY

p3a.endX + g3.width = p3b.startX p3a.endY + g3.height = p3b.startY

These are just a few of the thousands of constraints that are solved for a generated level. A description of how all of these constraints work together to produce a complete level, including a more detailed example showing constraints between beats, is provided in Section 5.1.

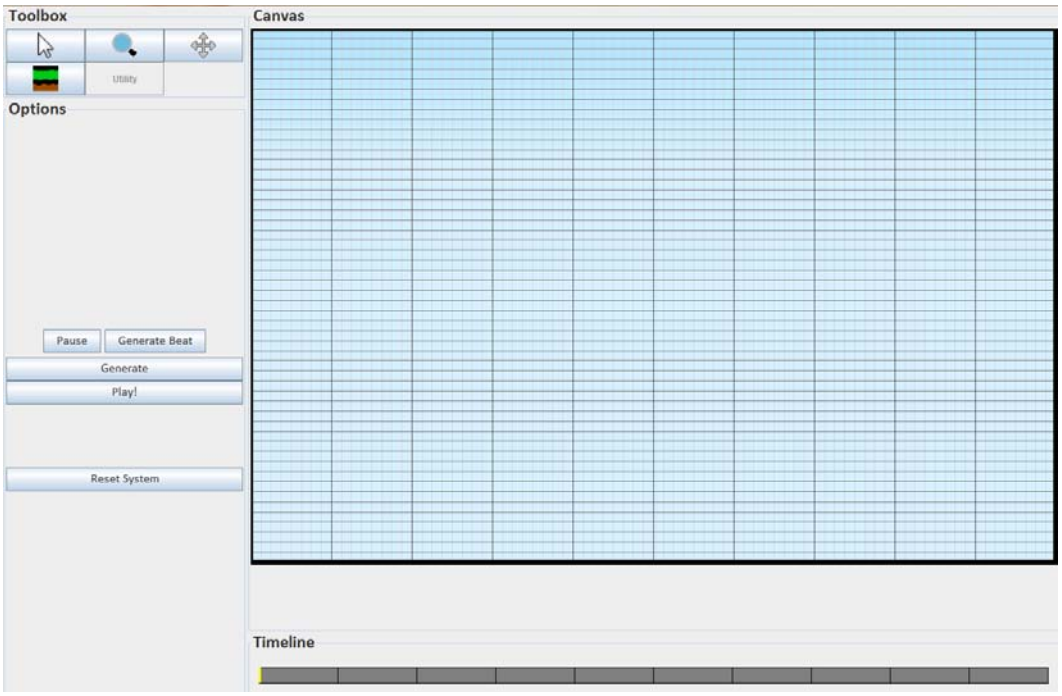

Figure 25. **The Tanagra design environment.** The level canvas in the top right is where the level is drawn by both the human and computer. A beat timeline along the bottom allows the designer to control level pacing independently from geometry. Controls on the left side allow the player to generate a level, playtest the level, and draw in geometry.

3 Design Environment

Figure 25 shows the Tanagra environment that designers are first presented with. The main region is the level canvas, a tile-based level editing tool that begins as an empty canvas[*]. Below the level canvas is the beat timeline, which is a long grey rectangle subdivided into smaller rectangles which represent beats. The width of each of the beat rectangles corresponds to the length of that beat. On the left of the screen is a toolbox containing controls for the designer, including the platform drawing tool. The "utility" pane in this toolbox contains three large buttons: to generate a new level, reset the level to its initial

[*] See Section 9 for a discussion of the interface changes made in a second version of Tanagra, in which the level canvas starts out populated with a generated level.

state, and playtest the level in the editor. Two smaller buttons above these allow the designer to temporarily pause the generator's response, and generate geometry for a single beat. Zoom controls for the level canvas and controls for pinning or unpinning geometry are also in the toolbox, as well as available in a context menu when right-clicking the level canvas. Platforms can be moved around the canvas by dragging them up and down. Beats can be added or deleted by right-clicking in the beat timeline, and beats are resized by dragging either the beat boundary markers or the platform anchors back and forth.

This section discusses the editing operations available to the designer for both geometry editing and beat creation and manipulation, and discusses when and why the generator runs in response to user changes.

3.1 Geometry Editing

The level canvas is the area where both the human and computer assistant draw and manipulate level geometry. The canvas is made up of a grid of tiles, scaled to fit into the window. The tile-based structure is primarily for ease of design, as tile-based level editors are extremely common for 2D games. However, this structure also provides a reduced search space for the constraint solver, as geometry constraints can be expressed in terms of tiles rather than pixels, resulting in smaller domains for each constraint variable.

The following geometry editing operations are available to the designer:

- **Draw Platforms.** A designer can place tiles into the level canvas just as she would for a non-intelligent level editor. When placing platform tiles, Tanagra automatically detects

the individual platforms that these tiles create, adding them to the appropriate beat for their position in the level canvas and creating a new beat if necessary.

- **Pin/Unpin Geometry**. Once tiles have been placed into the canvas, platforms can be selected and either “pinned” in place so that they stay where they are, or “unpinned” so that the generator can create new geometry in their place.

- **Move Geometry**. Platforms can be selected and moved up and down in the level canvas. When these platforms are moved, the remainder of the level morphs around the newly constrained platform, changing as little as possible. Platforms can be moved by dragging the platform anchor points that correspond to beat entry and exit.

3.2 Beat Editing

The beat timeline provides a mechanism for editing the pacing of the level by inserting or removing beats and modifying their length. Beat changes prompt Tanagra to make geometry changes, allowing pacing changes without the need to manipulate geometry. The beat timeline maintains a consistent overall length throughout the design process. The following beat timeline editing operations are available to the designer:

- **Resize Beat**. Change the length of the selected beat, automatically adjusting the length of its neighboring beats. The length change can be made at the beginning or end of the beat.

- **Split Beat**. Add a new beat to the level by splitting the selected beat in half. Any geometry contained in the original beat is retained and re-evaluated to fit the new

length constraints. Unless it contains user-placed tiles, the new beat has no geometry in it and can have geometry generated for it.

- **Remove Beat**. Remove the beat and its associated geometry, and change the length of its two neighboring beats to be adjacent to each other.

3.3 PLAYTESTING

Playtesting is crucial to the level design process. Typically a designer will frequently playtest a level to check for both its playability and how challenging it is, before giving the level to an internal Quality Assurance team or to external playtesters (Novak & Castillo 2008). Therefore, it is important for level design tools to support the designer in testing the level at any given time. A "play" button on the left side of the screen can be clicked at any time in the design process and puts the tool into playtesting mode. In this mode, the level canvas is replaced with an area where the designer can play the level. This allows the designer to quickly and easily test any changes made in the level. A yellow bar in the beat timeline shows the current position of the avatar, to aid in identifying problem areas in the level. Figure 26 shows the tool when it is in playtesting mode.

3.4 GENERATOR INVOCATION

The designer may request that the generator run at any time during the editing process by clicking the "Generate" button. This button re-generates level geometry, incorporating any user-pinned geometry. The designer may also request that geometry be re-created for the selected beat. If a new beat is created during editing and the level already contains

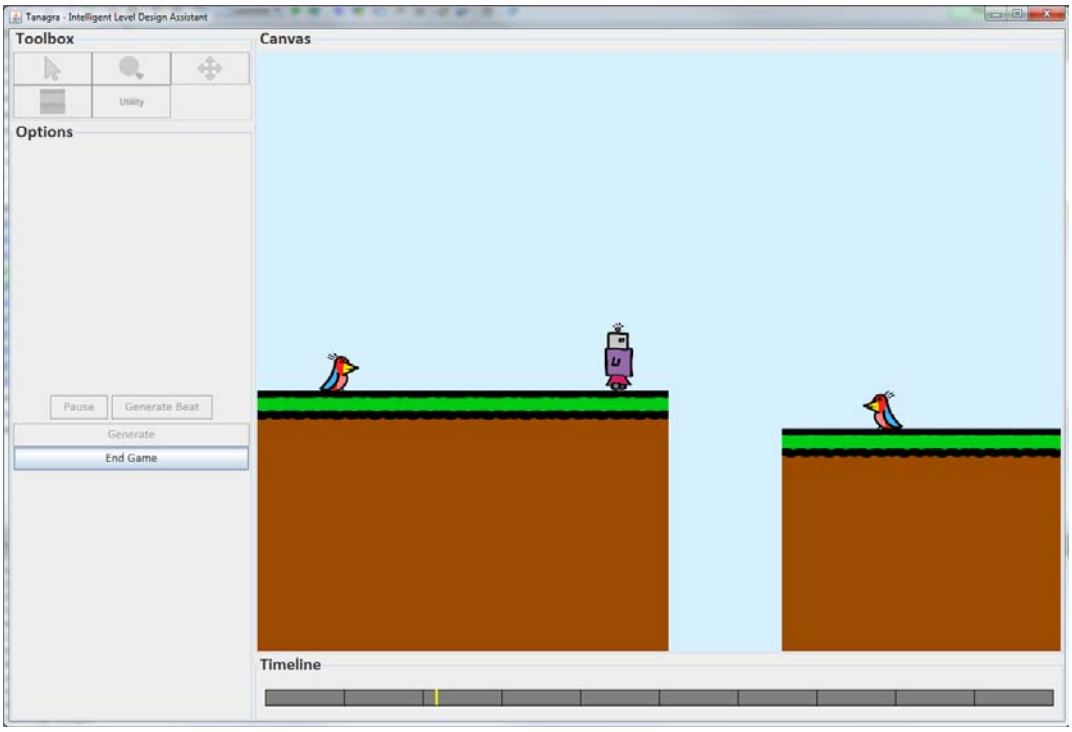

Figure 26. **The Tanagra tool in playtesting mode.** The avatar is represented by the robot; a yellow bar inside the beat timeline shows where in the level the avatar is located.

geometry, then the generator is called for that new beat to ensure that the level remains playable.

The level "re-solves" (i.e. the constraint solver is called again) whenever a change has been made that potentially alters the playability of the level. If a platform is moved then it is important to ensure that move did not make the level unplayable, e.g. if the platform movement caused a gap to be too high to cross. Re-solving the level can cause the generator to be invoked if no solution can be found by keeping all the geometry patterns in place. This use of the generator is discussed further in Section 7.

4 System Overview

Tanagra integrates reactive planning and numerical constraint solving to perform level generation and enable the editing operations described above. The level generator fulfills the following requirements:

1. Autonomously create levels in the absence of designer input.
2. Respond to designer input in the form of placing and moving geometry.
3. Respond to designer input in the form of modifying the beat timeline.
4. Ensure that all levels are playable.

Figure 27 is a general architecture diagram for Tanagra, showing how the components of the system interact with each other. As discussed earlier, we use the reactive planning language ABL (A Behavior Language) (Mateas & Stern 2002) to respond to designer input, choose the geometry that should be placed for each beat, and communicate with Choco. Choco (Choco Team 2008) is the constraint solving library used to specify and solve constraints on the

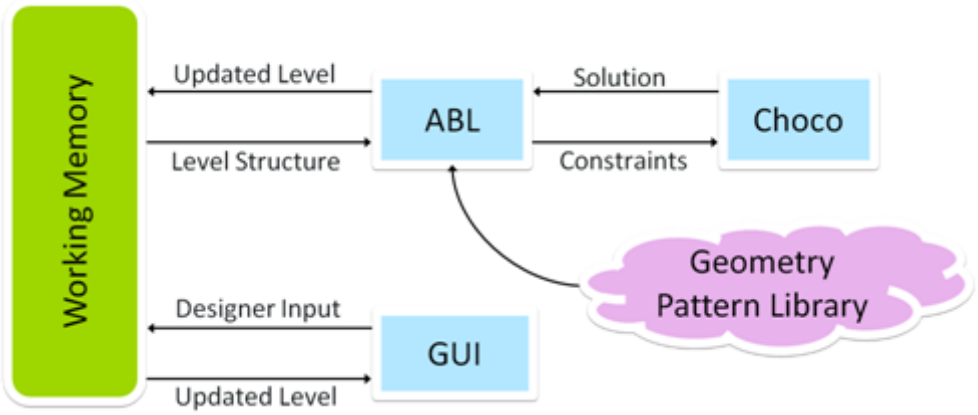

Figure 27. **Tanagra main architecture diagram.** Tanagra is made up of three main components: the GUI, an ABL agent, and the Choco constraint solver. The GUI and ABL communicate through working memory. ABL posts constraints to Choco and determines when the solver should be called; Choco responds with either a potential solution or a notification that no solution exists. A library of geometry patterns are specified using ABL behaviors.

placement of different level components.

The Tanagra ABL agent can be imagined as a subordinate assistant to the primary level designer. It can suggest different potential designs based on what has been done so far, and obeys the primary designer's commands for geometry or pacing changes. This ABL "assistant" has a colleague, Choco, who determines the precise physical placement of components in the level, and is responsible for reporting to the ABL agent if his changes lead to an unplayable level. If so, ABL and Choco then go back and forth attempting different configurations of level geometry until they find one that works to meet the designer's demands, or conclude together that there is no such level and report failure to the human designer.

Tanagra works quickly enough to permit rapid reaction to designer input. It can generate parts of levels in response to a change made by the designer, or can regenerate the entire level on demand, while respecting any constraints placed by the human designer. We have found it useful to keep separate the choice of a geometry pattern (using ABL) from the instantiation of that pattern (using Choco), as the precise placement of level components is influenced by surrounding geometry. This means that the placement of components in one beat may be able to change based on the placement of level components in a later beat, while still maintaining the same geometry pattern in both beats. Also, many different configurations of component placement meet the same geometry pattern. For example, a jump to a different platform could have a short initial platform and a long later platform, or vice versa.

The search for a valid level occurs in two stages: ABL searches at the structural, pattern level, and Choco searches for a valid, numerically-parameterized instantiation. The pattern abstraction also permits adding new kinds of design patterns easily, as instead of specifying all possible combinations of geometry components we can instead specify rules for the construction of the pattern.

This section provides an overview of how ABL and Choco work together, their responsibilities in Tanagra, and how they communicate with each other. Sections 5 and 6 provide a more detailed explanation of how different generation and editing operations are implemented.

4.1 Reactive Planning with ABL

ABL is a Java-based reactive planning language created by Mateas & Stern for use in creating believable agents that can rapidly react to a changing world state. Reactive planning, as a paradigm, is focused on domains where the world can change out from underneath an agent's own actions: this is exactly the situation in mixed-initiative level generation, where the designer makes manual edits that warrant a response from Tanagra. The world state is communicated using a blackboard architecture with Working Memory Elements (WMEs), which represent facts that the automated design agent knows and reasons about. These facts can be asserted or retracted from the agent's memory by either the human or the computer. The version of ABL used in Tanagra is one that I modified to support inheritance for WMEs. Every object in the world that the Tanagra ABL agent must interact with is stored as a WME: Figure 28 shows the inheritance hierarchy for all WMEs used in Tanagra.

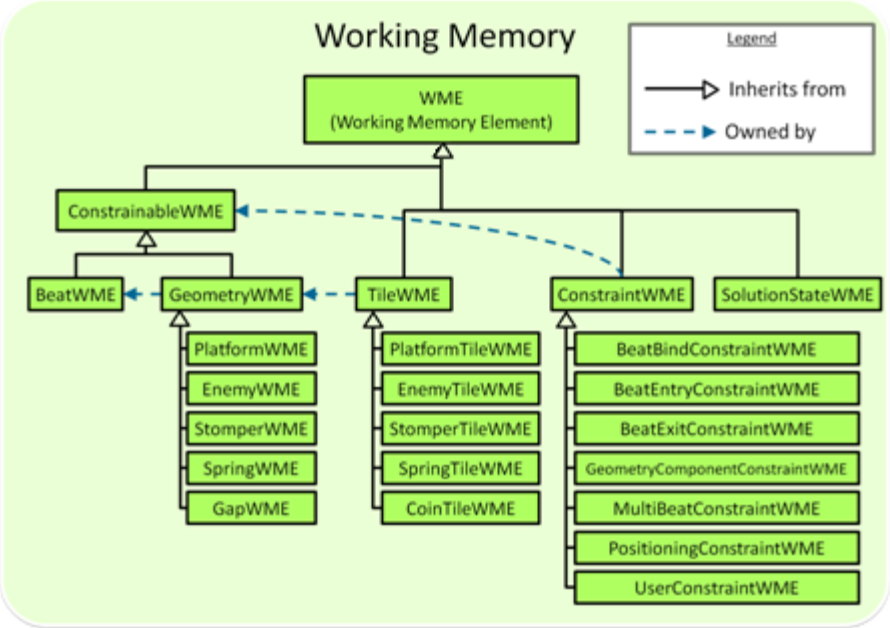

Figure 28. **Tanagra working memory knowledge representation hierarchy.** The hierarchy of working memory elements used to represent levels in Tanagra. Solid, black arrows denote inheritance; dashed blue arrows denote ownership. For example, *ConstraintWME*s have multiple owners, each of which must be a *ConstrainableWME*.

ABL agents are authored as a set of hierarchical *behaviors* that can be performed towards *goals*, where each of these behaviors can have a number of subgoals, similar to the behavior and goal relationship in a hierarchical task network. These behaviors can operate either in sequence or in parallel. Behaviors "ground out" in direct actions that should be taken in the world. In Tanagra's case, these actions are expressed in *mental acts*, which are written as normal, sequential Java code. There are typically several behaviors that can fulfill a particular goal. A particular behavior is chosen to fulfill the goal based on whether or not its

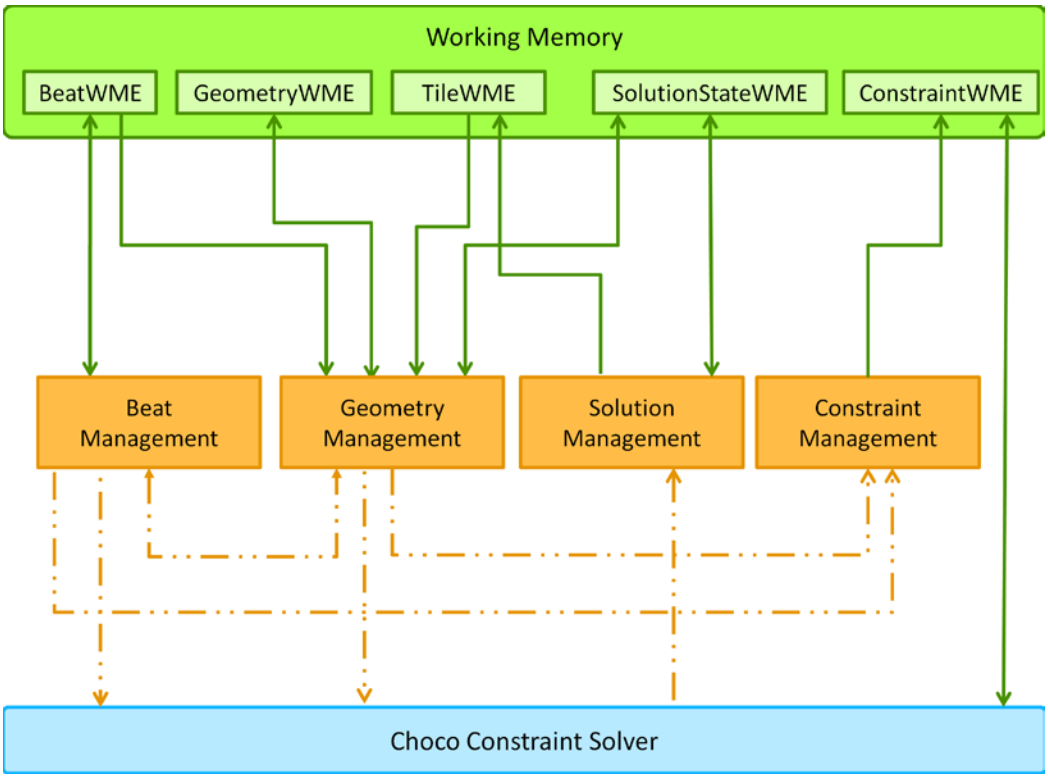

Figure 29. **Tanagra management and communication diagram.** This diagram shows the communication channels between working memory, ABL, and Choco, and dependencies between different managers within ABL. Green solid lines denote WME reading and writing, and orange dashed lines denote behavior and function calling.

precondition is true, and what internal priority (*specificity*) is given for it. If a behavior fails for any reason, the next most suitable behavior is chosen to fulfill the goal.[*]

Tanagra relies on a number of parallel behaviors called *managers* which wait for a change in a certain aspect of the world state and then initiate behaviors that react to that change. Much of the intelligence behind Tanagra comes from these managers working towards different, but related, goals in concert. For example, consider a scenario in which a designer adds a tile to an existing partial level. The ABL behavior that reacts to this change may add

[*] For a more complete description of ABL's semantics, inner workings, and design idioms I refer the reader to other literature on the topic (Mateas & Stern 2002; Weber et al. 2010).

or extend an existing piece of geometry, which in turn modifies the length of the beat containing the tile. The change also mandates the need for further constraints to be placed on the level. All these changes may also lead to a need for different geometry to be generated for the remainder of the level, to maintain its playability. Each of these scenarios is handled by separate managers, which are specified independently from each other.

There are four different categories of managers. Beat management and geometry management handle changes made to beats and geometry, respectively. In parallel, the solution managers address solution management and assignment, and the constraint managers handle adding constraints to the level as necessary. Each manager has a different priority; for example, beat management has a higher priority than geometry management, since the geometry that can be placed is dependent on its beat's length being predetermined. Figure 29 shows the communication model for Tanagra between working memory, ABL managers, and the Choco constraint solver.

4.2 Numerical Constraint Solving with Choco

Choco is a Java library for modeling and solving numerical constraint satisfaction problems. Constraints can be expressed on variables that are boolean, integer, or real numbers. These constraints are then read by Choco's constraint solver, which finds a value for each variable such that all constraints are satisfied.

We use Choco to model constraints for the geometric relationships within and between level components in geometry patterns (see Section 2.2.1), and also for the relationship between each geometry pattern and its associated beat. These constraints are designed to

ensure that the level is playable by taking into account a simple avatar physics model. Tanagra is aware of the player's movement speed, initial jump velocity, and the impact that springs have on this jump velocity. All variables associated with a level component have a domain: the range of potential integer values that the variable can potentially be solved as. Positioning variables for platforms, enemies, and springs have a domain determined by the size of the level canvas. Gaps have a domain determined by the physics model—their width and height are not allowed to exceed the maximum distance the avatar can jump. Therefore, Choco is only allowed to choose values for variables from a range that guarantees playability. The constraints placed on these variables by other aspects of level generator and user editing further winnow down the space of potential solutions.

All constraints store a set of their "owners", which are all *ConstrainableWME*s (i.e. beats and geometry components). This ownership property makes it straightforward to remove constraints whenever geometry or beats are modified or deleted. As seen in Figure 28, there are a number of different kinds of constraints, each of which serves a different purpose in ensuring the playability of levels. Constraints are differentiated to ensure that some of them (such as those that specify internal geometry constraints) are never retracted during solving, and others (such as those that force geometry to a specific position) are retracted at the appropriate time. The *ConstraintWME*s below encode facts about the structure of the level:

- **BeatBindConstraintWME.** This is a constraint that binds the exit point of one beat to the entry point of the next beat, and vice versa. It ensures that geometry between beats "lines up".

- **BeatEntryConstraintWME, BeatExitConstraintWME.** These constraints bind the start point of the entry platform for a beat to the entry point of that beat, and the end point of the exit platform for a beat to the exit point of that beat.

- **GeometryComponentConstraintWME.** This is a constraint internal to specific geometry. For example, the expression that the width of a platform is equal to its ending x position minus its starting x position.

- **MultibeatConstraintWME.** This is a constraint that is applied to geometry within patterns that span multiple beats. For example, the staircase pattern consists of three gaps between platforms. The multibeat constraints for these gaps ensure that each gap has identical width and height.

- **PositioningConstraintWME.** This is a constraint that is placed on every geometry component when a solution is found, that forces geometry to stay in the same position whenever possible. Positioning constraints are frequently added and removed during the geometry search process.

- **UserConstraintWME.** A constraint placed on geometry by the user, either when drawing platforms into the canvas or by pinning geometry in place. User constraints are never removed unless the designer unpins or removes the constraints' owner.

Choco is called from ABL each time the generator has finished placing geometry for all beats; typically, solutions are found within 5 milliseconds, on a 3GHz Intel Core 2 Duo. It can take significantly longer to exhaustively determine that there is no solution, so Tanagra

stops Choco from searching after 50 milliseconds, since it is unlikely to find a solution after that point. For each variable, Choco is set to attempt a random integer value in its domain (after removing known values that it cannot be based on constraint propagation), allowing it to create a variety of levels even when using the same geometry patterns.

5 Geometry Management and Generation

The majority of ABL behaviors in Tanagra deal with managing user-provided geometry and generating new geometry. These behaviors can be grouped into three major categories: expression and implementation of hierarchical geometry patterns (similar to grammar production rules), creating level components from user-placed tiles, and incorporating user-created geometry into patterns. This section will discuss in detail how ABL and Choco interact to perform these tasks.

5.1 Hierarchical Geometry Patterns

At its base, a geometry pattern consists of a set of level components and constraints that are asserted on these components. Geometry patterns can contain other geometry patterns, and optionally additional constraints on components that exist between patterns. Once a pattern has been assigned to a beat, the solver ensures that at least one instantiation of the pattern still allows the level to be playable.

Driving the generation of patterns is the `MissingGeometryManager`, a persistent ABL behavior that waits to see if a beat is in need of geometry, and if it is, subgoals geometry generation for that particular beat. The code listing for this manager, along with behaviors for creating gap and staircase patterns is provided in Listing 1.

```
 1| sequential behavior MissingGeometryManager() {
 2|    BeatWME beat;
 3|    with (success_test {
 4|            beat = (BeatWME readyForGeometry==true
 5|                           generateRequested==true)
 6|            sWME = (SolutionStateWME
 7|                           hasRemainingSolutionStates==true)
 8|            (sWME.getCurrentStateAtBeat(beat) == NONE)})
 9|       wait;
10|    subgoal GenerateGeometry(beat);
11| }
12|
13| sequential behavior GenerateGeometry(BeatWME beat1) {
14|    precondition {
15|        beat2 = (BeatWME prev==beat1)
16|        beat3 = (BeatWME prev==beat2)
17|    }
18|    subgoal GenerateGapPattern(beat);
19|    subgoal GenerateGapPattern(beat2);
20|    subgoal GenerateGapPattern(beat3);
21|    subgoal CreateStaircaseConstraints(beat1, beat2, beat3);
22| }
23|
24| sequential behavior GenerateGeometry(BeatWME beat) {
25|    subgoal GenerateGapPattern(beat);
26| }
27|
28| sequential behavior GenerateGapPattern(BeatWME beat) {
29|    precondition {
30|        solutionState = (SolutionStateWME)
31|        (solutionState.getCurrentStateAtBeat(beat) == NONE)
32|        (solutionState.checkState(beat, GAP))
33|        !(PlatformWME owner==beat isEntry==true isExit==true)
34|    }
35|    mental_act {
36|        solutionState.updateState(beat, GAP);
37|    }
38|    subgoal placePlatform(beat, ENTRY);
39|    subgoal placePlatform(beat, EXIT);
40|    subgoal placeGap(beat);
41|    subgoal createGapPatternConstraints(beat);
42| }
43|
44| sequential behavior GenerateGapPattern(BeatWME beat) {
45|    precondition {
46|        solutionState = (SolutionStateWME)
47|        (solutionState.getCurrentStateAtBeat(beat) == GAP)
48|    }
49|    succeed_step;
50| }
```

Listing 1. **A sampling of the geometry pattern ABL behaviors written for Tanagra.** This listing includes partial code for checking if beats need geometry, and generating staircase and gap patterns.

This listing shows the hierarchical nature of geometry patterns, and the similarities between grammar production rules and the ABL behaviors. The manager on lines 1-11 is responsible for checking to see if there are any beats with missing geometry, and if so dispatch instructions to fill the beat with geometry. There are multiple versions of the `GenerateGeometry` behavior, each of which corresponds to a different production rule; two of these behaviors are shown in this example. ABL will randomly select one of these behaviors (that passes its precondition check) at runtime to fill the beat with geometry; if one of those behaviors fails for any reason, it will try other behaviors until it either finds a successful way to fulfill the goal or determines that no behavior can do so.

The first `GenerateGeometry` behavior, on lines 13-22, checks to see if there are two subsequent beats available in the level, and if so, creates a staircase pattern across all three beats by creating gap patterns in them and then placing additional staircase constraints, as described in Section 2.2.1. Notice that there are two versions of `GenerateGapPattern` (lines 28-42 and lines 44-50); the first creates a new gap pattern if there is no pattern in the beat already, the second succeeds without taking any further actions if there is a gap pattern in the beat. Failure is implicit—if there is a pattern in the beat that is not a gap, then there is no behavior to handle this situation, and the goal of placing the gap pattern into the beat will fail, propagating backwards and forcing ABL to choose a different pattern to place in the beat. The second `GenerateGeometry` behavior, on lines 24-26, is a simple single-beat pattern for placing a gap.

When placing platforms into the level, as in lines 38 and 39 of the listing, they must be flagged as either entry or exit platforms. Tanagra's representation for levels mandates that

no beat may contain more than two platforms, since each beat may contain only a single player action, such as jumping or waiting. An enemy or a stomper pattern uses a single platform; a gap or spring pattern uses two platforms. Playability is guaranteed by constraining the exit platform of one beat to have an identical vertical coordinate to the entry platform of its neighboring beat; these constraints are handled during beat management—whenever a beat is added or removed, the constraints are updated. Note that while the level may still be playable if the platforms had positions that differed by only one or two tiles, this difference would constitute a separate player action (i.e. a jump between platforms) which would belong in a separate beat.

Figure 30 shows a diagram representing a partially specified level; Table 2 describes its constraints. In this example, the first three beats in the level are assigned the valley pattern, and the final beat is assigned the spring pattern. The valley pattern consists of a jump down over a gap, followed by a platform with an enemy on it, and finally a jump over a gap, consuming three beats in total. The spring pattern consists of two platforms, a spring at the end of the first platform, and a tall gap between the two platforms. The following list explains what each of the numbered constraints are and why they are placed in the system:

1. This constraint is placed during the beat management phase of content generation; beats A and B are constrained such that the exit point of beat A is equal to the entry point of beat B. This ensures that the level will be playable.
2. Platform E is an exit platform from beat A; this constraint requires that the end position of the platform is tied to the exit point of the beat. This constraint is placed when the platform is created.

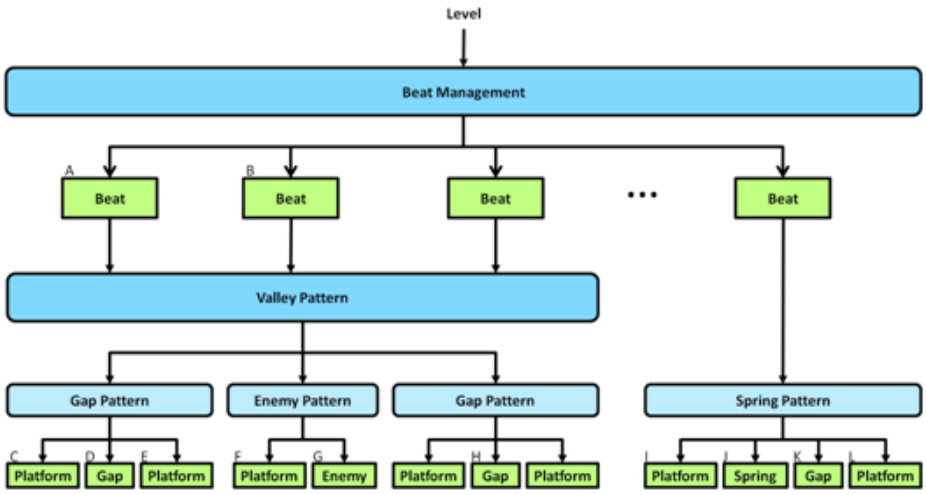

Figure 30. **A diagram representing a potential partial level during level generation.** Beats A and B are consumed by the Valley Pattern, which in turn is a gap pattern, enemy pattern, and another gap pattern. The final beat in the level is assigned a spring pattern, which places two platforms, a spring, and a gap. Blue, rounded boxes denote ABL behaviors. Green boxes are WMEs. Not represented here are the tiles that make up each level component. Table 2 provides examples of the constraints imposed between each level component.

Number	Cause	Constrained	Constraint equation
1	Beat Management	A, B	A.exitPointX = B.entryPointX A.exitPointY = B.entryPointY
2	Beat Management Gap Pattern (1)	A, E	A.exitPointX = E.endX A.exitPointY = E.endY
3	Beat Management Enemy Pattern	B, F	B.entryPointX = F.startX B.entryPointY = F.startY
4	Beat Management Gap Pattern (1)	A, C, D, E	C.length + D.length + E.length = A.length
5	Geometry Component Constraint	C	C.startX + C.width = C.endX
6	Valley Pattern	D, H	D.width = H.width D.height = -1*H.height D.height > 0
7	Gap Pattern (1)	C, D, E	C.endX + D.width = E.startX C.endY + D.height = E.startY
8	Enemy Pattern	F, G	G.posY = F.posY – 1 F.startX <= G.posX <= F.endX
9 10	Spring Pattern	I, K, L I, J	I.endX + K.width = L.startX J.posY = I.posY – 1 J.posX = I.endX
11	J (Spring)	K	K domain change due to J's physics modification

Table 2. **Example constraints on beats and geometry in a partial level.** A representative sample of the constraints imposed on the example level structure shown in Figure 30.

3. Similarly, platform F is created as both an entry and exit platform for beat B, as part of creating the platform in the enemy pattern behavior. Therefore, F's start position is tied to the entry point of beat B.
4. To ensure that all geometry components fit into a single beat's duration, a constraint such as this is used: the sum of the lengths of each component in the beat must be equal to the beat's length. Here, platform C, gap D, and platform E are constrained to fit within beat A.
5. There are many geometry component constraints that are used to specify internal constraints that should never change; these constraints are added to the system when the associated level component is created, and removed when the level component is deleted. One example is that the start position of platform C plus its width must be equal to its end position.
6. These constraints are all multibeat constraints placed during the creation of a valley pattern. The width of gaps D and H must be equal to each other, and their heights must be exactly opposite of each other. Gap D's height must be greater than zero (the top-left corner of the level canvas is point (0,0)) in order for it to be a valley rather than a mesa.
7. This constraint is identical to the one described in Section 2.2.1 for the gap pattern: a gap's width and height constrains the start and end positions of the other platforms in the beat.
8. Enemy patterns require that there be an enemy placed on top of the beat's platform; these constraints ensure that the Y position of the enemy is exactly one tile above the platform in the beat, and that its X position is somewhere between the start and end of the platform.

9. Similar to constraint number 7, this constraint states that the height and width of the gap constrain the relative positions of its surrounding platforms.
10. These two constraints correspond to the position of the spring on the entry platform of the beat. The spring must be a single tile above the platform's position in Y, and be placed at the rightmost tip of the platform in X.
11. Finally, to continue to ensure level playability, the creation of spring J in front of gap K results in a variable domain change for K's height: it can now be much taller than if there were no spring there at all.

These constraints form only a small, representative sample of all of the constraints in a level. A complete level has thousands of constraint equations that Choco solves for, often multiple times in order to find a solution that minimizes changes made to the level (see Section 7).

5.2 Creating User Geometry from Tiles

The designer interacts with the level canvas by placing individual tiles into it. Since all level generation and playability verification is performed on level components rather than individual tiles, Tanagra must determine the appropriate level components based on the tiles that it detects such that Tanagra's internal representation for levels is not violated. This geometry is then incorporated into geometry patterns as appropriate.

Platforms, like any other geometry, are "owned" by a particular beat. While handling platform placement from tiles, it is also important to determine if the placed platform is an entry platform, exit platform, both, or neither, to assist with the pattern assignment

described above. Also, constraints must be added to the system to ensure that the newly created platform stays in the place indicated by the designer.

This entire process of creating platforms from tiles is accomplished in two stages, using two different managers: the first, higher priority, manager is responsible for constructing platforms from tiles, and the second is responsible for placing constraints on the platform and determining entrance and exit properties. When creating a platform from tiles, ABL determines if the tile belongs to a platform that isn't already in working memory, should be added to an existing platform, or should cause the merging of two neighboring platforms. Beat ownership and entry/exit determination are calculated by the position of the tile: from this position, and an understanding of the movement speed of the avatar, we can calculate the beat that the platform must belong to. A newly created platform is given constraints such that the tiles placed by the designer will always be contained in the platform, but the platform may eventually contain additional tiles. This allows platforms to extend to fit whichever geometry pattern they are assigned to.

5.3 Incorporating User-Created Geometry into Patterns

A crucial feature of Tanagra is its ability to incorporate user-created geometry into patterns during geometry generation. This is part of what constitutes the "reactive" aspect of the reactive grammar we created with ABL and Choco. Because constraints are expressed on individual geometry components and beats, rather than on the pattern as a whole, it is possible to simply swap out an existing, user-created platform for a platform that otherwise would have been generated, and continue to have additional constraints expressed as normal.

```
01| sequential behavior placePlatform(BeatWME beat, boolean entry,
02|                                   boolean exit) {
03|    precondition {
04|        plat = (PlatformWME owner==beat isEntry==entry
05|                                        isExit==exit)
06|    }
07|    mental_act {
08|        beat.addGeometry(plat);         //add geometry to beat
09|        plat.setReadyForSolving(true);  //platform ready to solve
10|    }
11| }
12|
13| sequential behavior placePlatform(BeatWME beat, boolean entry,
14|                                                  boolean exit)
15| {
16|    specificity -1;  //choose this behavior last
17|    precondition {
18|        !(PlatformWME owner==beat isEntry==entry isExit==exit)
19|        (beat.numPlatforms() < 2)
20|    }
21|    mental_act {
22|        //create a platform with desired properties & add to beat
23|        PlatformWME plat = new PlatforMWME(beat, entry, exit);
24|        beat.addGeometry(plat);
25|
26|        //add the new platform to working memory
27|        BehavingEntity.getBehavingEntity().addWME(plat);
28|    }
29| }
```

Listing 2. **Platform placement ABL behaviors.** Two different behavior definitions for placing a platform into a specific beat, given that the platform must have the specified entrance and exit properties in relation to the beat.

Recall from Listing 1 that the geometry generation behaviors subgoal actions such as `placePlatform` (Listing 1, lines 38-39). Listing 2 shows the ABL code for two of the five different `placePlatform` behaviors used in Tanagra's geometry creation. The first of these behaviors (lines 1-11) handles the situation in which there is already a `PlatformWME` whose beat owner, entry, and exit conditions exactly match those specified in the behavior's arguments. This platform is bound to the variable `plat` (lines 4-5). If the precondition succeeds, then `plat` is added to the beat's geometry set and is flagged to be ready for solving by Choco (lines 7-10). If this behavior's precondition fails, then it is still possible for the second behavior (lines 13-27) to execute for the goal. The second behavior

handles the case where there is no `PlatformWME` that fulfills the stated requirements, and less than two platforms already in the beat (lines 17-20). In this case, a new platform is created (which, by default, is ready for solving) and added to working memory (lines 21-28).

Currently, the only supported user-created geometry in Tanagra is platforms; other level components are placed into patterns by the generator using behaviors similar to the second one in the above example.

6 Beat Management

There are three main design concerns that Tanagra must be able to reason about with regard to the beat timeline: handling responses to the designer making a change to the length of a beat, splitting a beat to create a new one, and deleting a beat. Each of these are resolved by separate ABL managers. These managers work by monitoring for a flag that is set on a beat when the designer initiates one of these changes. When the flag appears, a new behavior is subgoaled to handle the necessary changes. In the case of length changes, the behavior updates any relevant constraints for beat entry and exit positions; because platform positions are tied to the beat entry and exit positions, there is no need to update any constraints on geometry components. When deleting a beat, the behavior ensures that all geometry and associated constraints are removed from the system, and ties together the beat's preceding and subsequent beats with new beat constraints. When adding a beat, it is flagged as being ready for geometry; doing so ensures that the `MissingGeometryManager` described in Section 5.1 will see the new beat and place new geometry into the level.

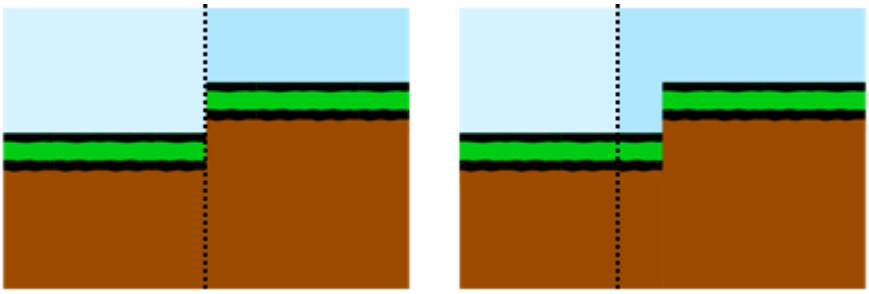

Figure 31. **Correcting beat placement.** This figure illustrates the necessity for correcting beat placement based on geometry placement. The blue shaded boxes and dashed lines denote beat boundaries. If the designer draws geometry as depicted on the left, there can never be a solution to the level as the platforms do not line up at the beat boundary. However, this geometry placement should clearly be playable. The right side shows how Tanagra corrects beat boundaries to address this issue.

There are also two managers that handle the relationship between user-placed platform tiles and beats. The `TooManyPlatformsManager` handles a slightly more subtle problem that can arise from a designer placing geometry into the level canvas. Tanagra begins with a default number of beats in the timeline, but the designer is capable of freely drawing platforms wherever (and however frequently) she wishes. Recall, however, that a single beat is only ever allowed to contain at most two platforms, which are the beat's entry and exit platforms. Therefore, when a designer places more than two platforms into a beat, Tanagra must respond by splitting beats in an appropriate location. An associated manager, the `PlatformMatchupManager`, handles a related problem: if the designer draws two platforms at different vertical positions, but whose end-points match up to the same beat boundary, Tanagra should be able to move the beat boundary in attempting to ensure level playability (Figure 31). These two managers constantly monitor for platforms that violate these rules, and raise the beat length change flag or beat split flag as appropriate.

7 Constraint Solving and Search

The motivating force behind Tanagra's solving and search process is to minimize the number of global changes that must be made to a level in order to resolve small, locally made changes from either the human or computer designer. This is essential to providing the designer with an editing environment in which the response to any changes made to the level continue to ensure playability while leaving the level as similar as possible to its prior state. However, the designer should still be able to request a complete re-generation of the level at any time, in order to see different potential levels given the current constraints. This motivation leads to a two-stage solution and search process whenever a change is made to

Figure 32. **The necessity for constraint relaxation.** This example shows how constraint relaxation is necessary when creating levels. On the left, a level with a user-specified platform on the right is being generated. A gap pattern has been chosen for the first beat, and Choco has instantiated this pattern as a jump down. In the middle example, the second beat also has a gap pattern, but using the existing constraints for the first beat, the level is unplayable. The right example shows how relaxing the positioning constraints for the first beat renders the level playable, without changing any of the geometry patterns.

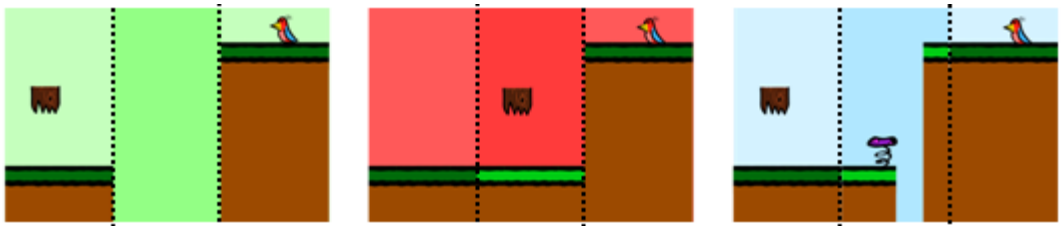

Figure 33. **The necessity for geometry search.** A small example illustrating the need for geometry search. On the left, a partial level is being solved, where existing geometry in the first and third beats is pinned in place by the user. A geometry pattern must be selected for the middle beat. In the middle level, Tanagra selected the stomper pattern to fill the middle beat, which leads to an unsolvable level, as the exit platform and entry platforms for beats two and three can never line up. On the right, Tanagra has instead selected the spring pattern, which leads to a playable level segment.

the level. The first stage iteratively relaxes constraints on the level; if necessary, the second stage iteratively re-generates small sections of the level until a solution is found.

Recall that ABL assigns positioning constraints to level components upon solving, to minimize the changes made to the level when new geometry is generated. These constraints must often be partially relaxed when new geometry is added (Figure 32). There are also many situations in which the first geometry pattern selected for a beat would be invalid. For example, consider a scenario in which the designer has pinned two long platforms that are separated by a single beat. These platforms have different y values, so the connecting geometry for the middle beat could not be another single platform, as the endpoints would not line up. However, using a different geometry pattern, such as a spring, may complete the level geometry to make a playable level. Figure 33 illustrates this scenario.

These two scenarios lead to the need for multi-stage constraint solving on Choco's side, and backtracking on ABL's side.

7.1 Constraint Solving

ABL calls the constraint solver by subgoaling the `Solve` behavior. This behavior takes as a parameter a list of ConstrainableWMEs whose positioning constraints should be ignored, called an "ignore set". `Solve` is subgoaled after each attempt to place a new geometry pattern during geometry generation, and after modifying the beat time through adding, deleting, or changing the length of a beat. The list of ConstrainableWMEs initially consists only of the new geometry components added to the level, or the beats affected by the timeline change. The `Solve` behavior calls a custom solver written in Java that iterates over

every ConstraintWME in working memory, excluding only positioning constraints belonging to objects in the ignore set. If a solution is not found, then positioning constraints are slowly relaxed outward from the original ignore set. On each attempt to solve, the solver finds the ignore set's "frontier" beats (i.e. those that are in the set but whose neighbors are not) and randomly chooses one of these frontier beats to add to the ignore set then re-solves. Constraints imposed on geometry by the user are never added to the ignore set. When the frontier meets user-specified geometry or the edge of the level, there is no solution for this particular configuration of geometry, and ABL must search for a solution. By running the solver so many times just for a single geometry pattern placement, this algorithm does incur a high solving cost, but provides the benefit of minimizing global changes. Even with the solver running so frequently in the worst case scenario, Tanagra still can find valid levels at an interactive rate.

7.2 Searching for a Solution

If the solver does not find a solution, then we can be sure that the last change made to the level has resulted in making it unplayable. There are two potential causes for this:

1) ***The concrete geometry corresponding to the most recent change leads to the level being unplayable, and should be replaced.***

To address this issue, Tanagra maintains a solution state. This state consists of the geometry patterns that are being used at any given time, indexed by the beat they belong to, and a set of states that are known to be invalid. Whenever the solver runs, a *CleanupSolver* behavior is subgoaled. This behavior can do four different things, depending on the solution state. If

the solver succeeds, then the *CleanupSolver* behavior simply records a success. However, if the solver fails, then the behavior must record the state currently being attempted as invalid and all non-user geometry and constraints should be removed from the beat. If there are remaining geometry patterns to be tried for a beat, then the cleanup behavior is complete, and the *MissingGeometryManager* will take care of attempting a new pattern. If there are no remaining geometry patterns for the beat, then non-user geometry and constraints are removed from both that particular beat and its neighbors, as the solver's failure signifies a more global problem with geometry placement. This search continues until either a solution is found, or it is determined that the failure to solve is due to a different cause:

2) ***The player has imposed constraints that contradict each other, such as pinning geometry combinations that can never lead to a solvable level.***

Due to the requirement that Tanagra respect every constraint that the designer places on a level, it is possible that the designer will place constraints that conflict with each other. This is most common during geometry pinning, by placing platforms which, no matter the intervening geometry, would never be reachable from each other given the game's mechanics. In this scenario, Tanagra changes the background of the level canvas from pale blue to red, denoting the lack of a solution. From here, the designer can begin manually removing offending geometry.

8 USE SCENARIO

This section presents a detailed use scenario, showcasing Tanagra's key abilities—auto-filling geometry, brainstorming level ideas, and manipulating level pacing—by showing

screenshots taken at key moments during a hypothetical designer's interactions with the system. There is also a video showing Tanagra in action[*].

Tanagra can generate levels with or without initial designer guidance. Figure 34 shows an example level generated without any input, using the default beat timeline, requested by the designer while she is searching for inspiration for what kind of level to create. Figure 35 shows a different potential beginning to level generation, where the designer has partially specified known desirable geometry; notice that this automatically altered the beat timeline to accommodate the extra platforms and corresponding extra number of actions the player would need to take.

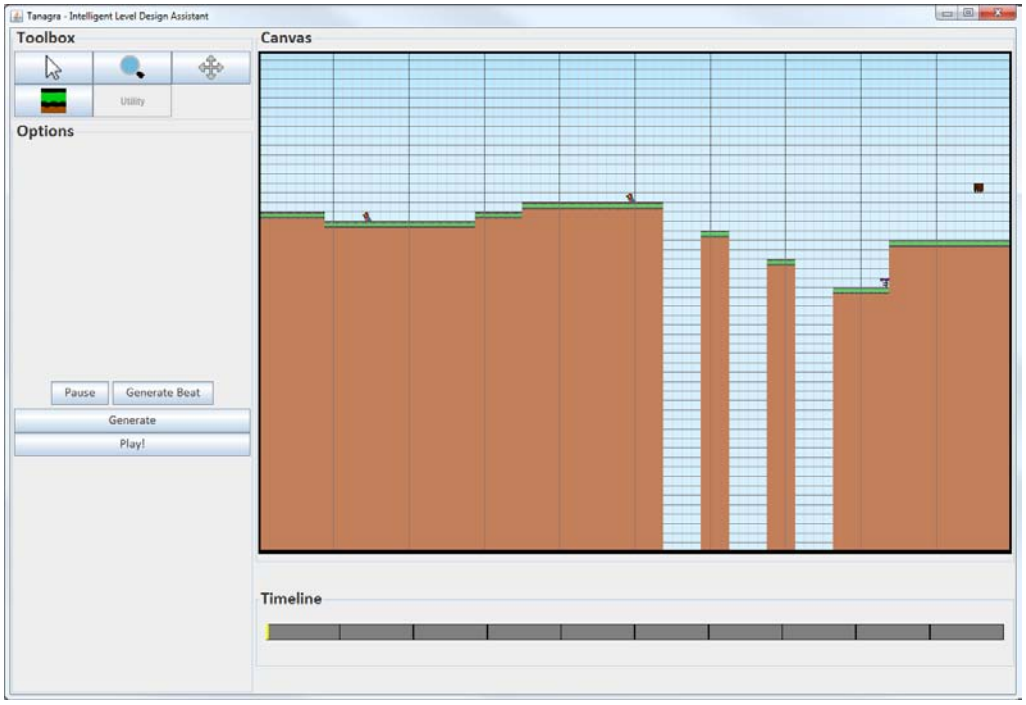

Figure 34. **Tanagra use case: full level generation.** An initial generated level provided by Tanagra.

[*] The video is available in this dissertation's supplemental files, and also online: http://sokath.com/dissertation_supplement/tanagra/TanagraMovie.wmv

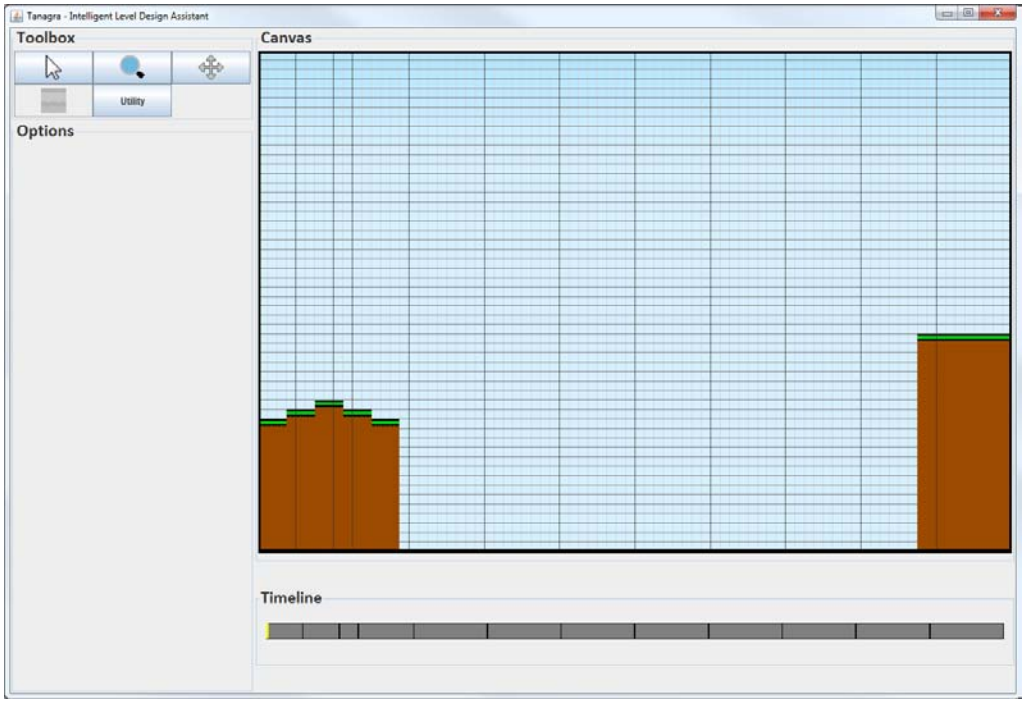

Figure 35. **Tanagra use case: user-drawn geometry.** Tanagra has automatically split the early beats in the level to accommodate the extra platforms.

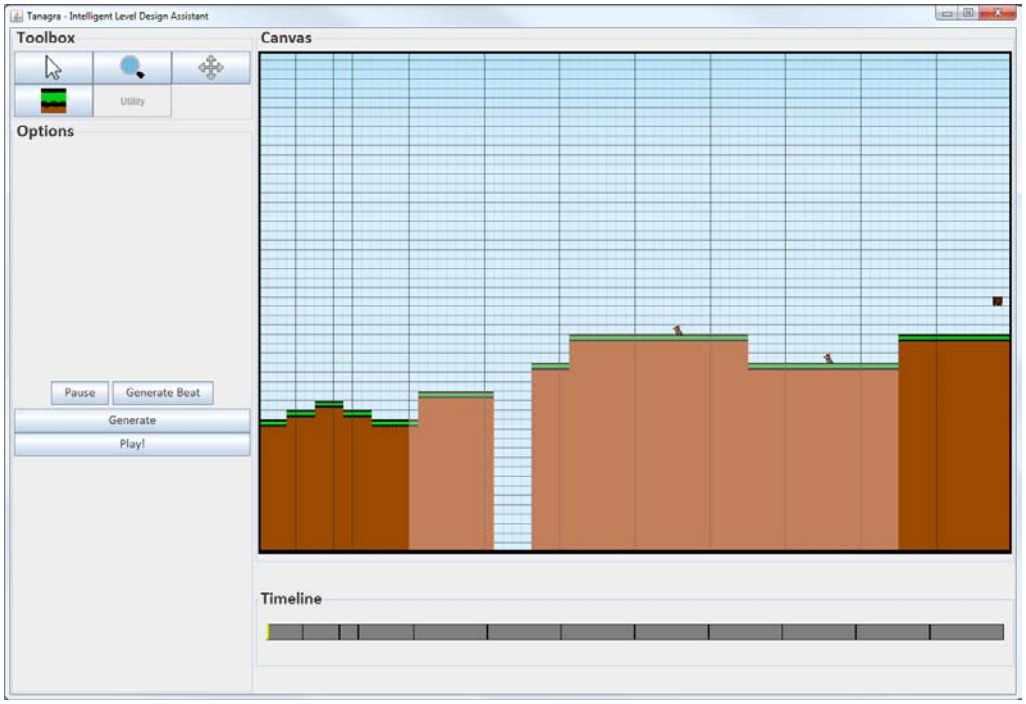

Figure 36. **Tanagra use case: auto-filling geometry (example 1).** The designer has asked Tanagra to auto-fill the remainder of the level with generated geometry. User-drawn and pinned platforms are shown darker than generated platforms.

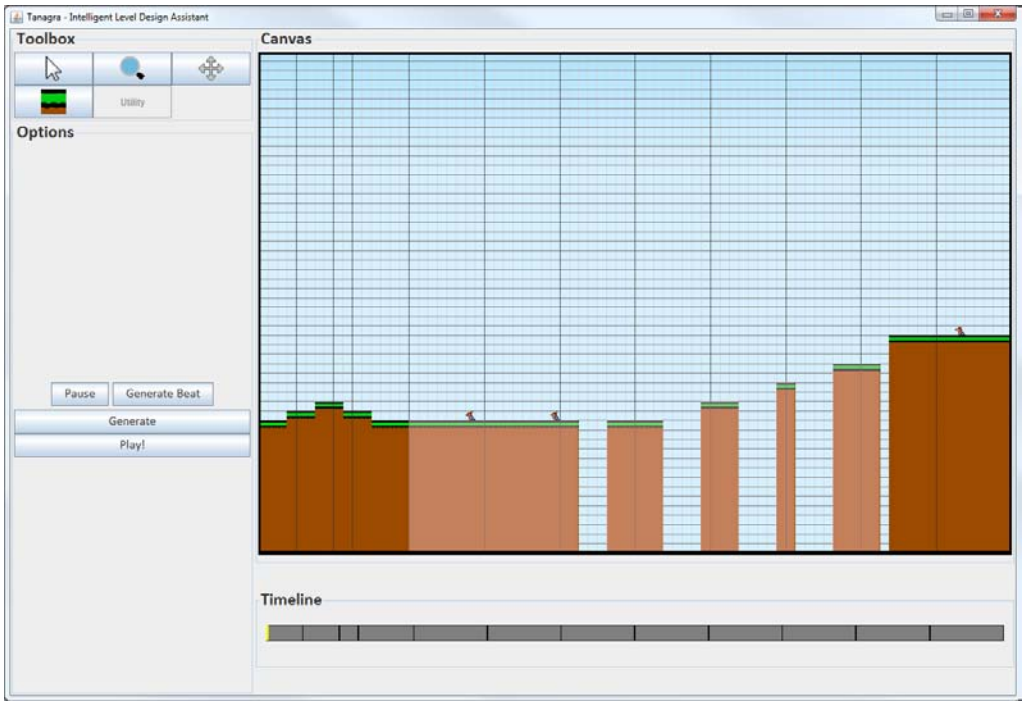

Figure 37. **Tanagra use case: auto-filling geometry (example 2).** A second potential level that fits the user-specified geometry.

Seeking further inspiration, the designer requests suggestions for how to fill the space between the two sections of geometry she has specified. Tanagra can rapidly regenerate the level to show different variations that meet the same requirements (Figure 36 and Figure 37). The designer can then play with the levels that have been generated, prototyping different design ideas. Figure 38 shows a scenario in which the designer has decided that the closing platform should be moved significantly higher; the remainder of the level is adjusted to ensure the level remains playable after this change has been made.

In Figure 39, the designer is prototyping different pacing ideas by significantly changing the pacing of the level. She makes the level more fast-paced in the middle by adding a number of beats, and slower paced at the beginning and end of the level by deleting beats there.

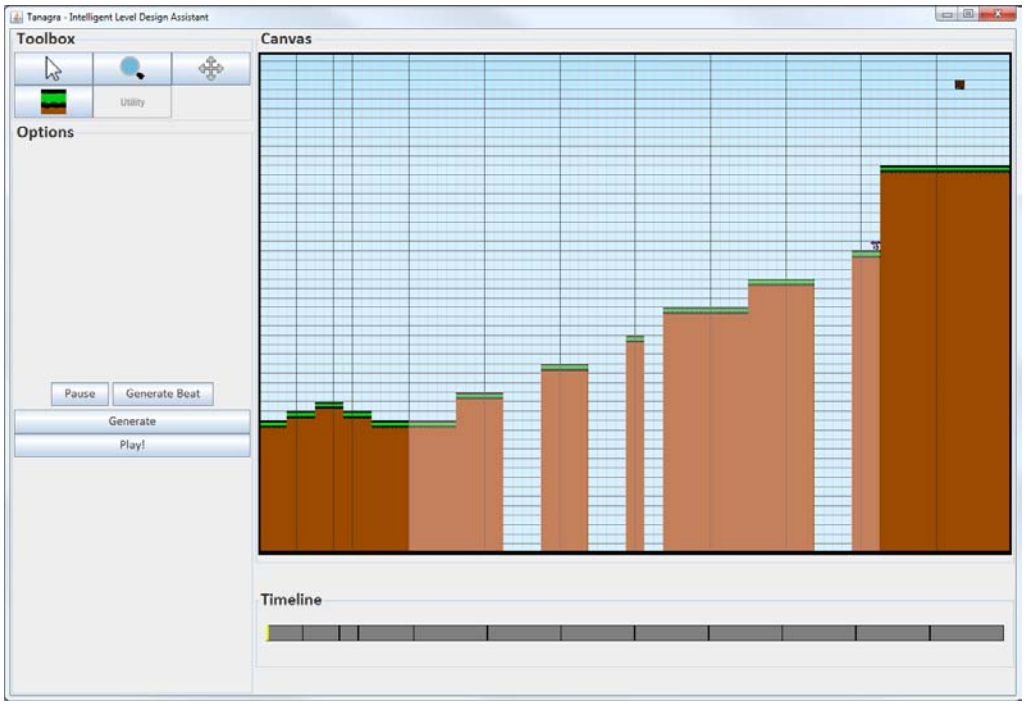

Figure 38. **Tanagra use case: moving geometry.** The designer moves the end platforms higher in the level.

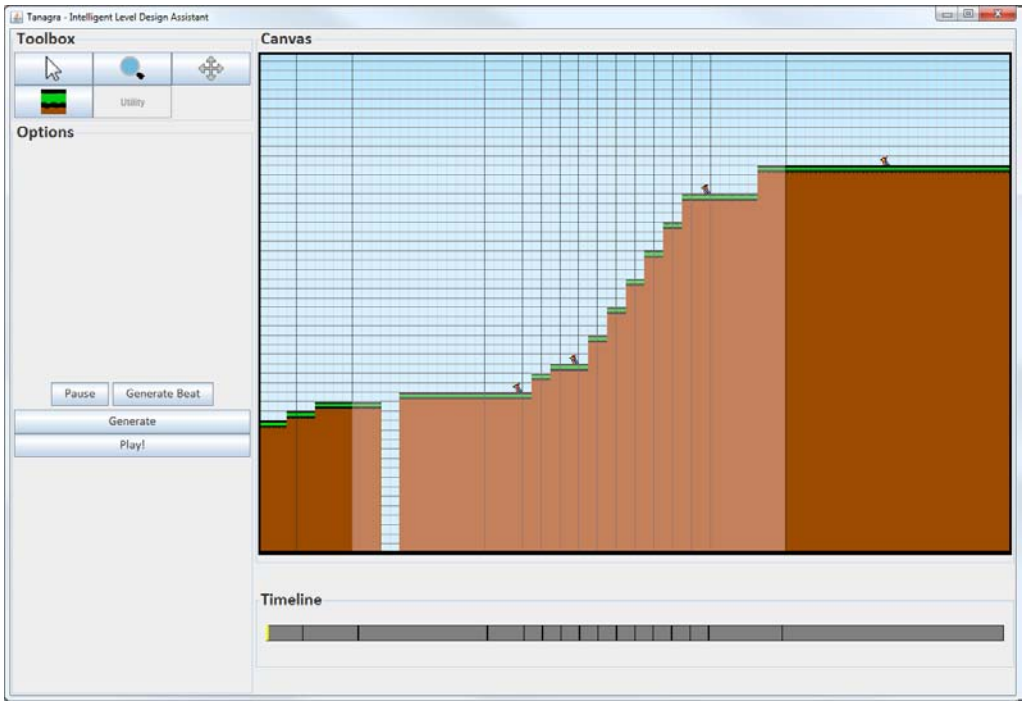

Figure 39. **Tanagra use case: rhythm changes (example 1)**. There are more beats in the middle and fewer at the beginning and end.

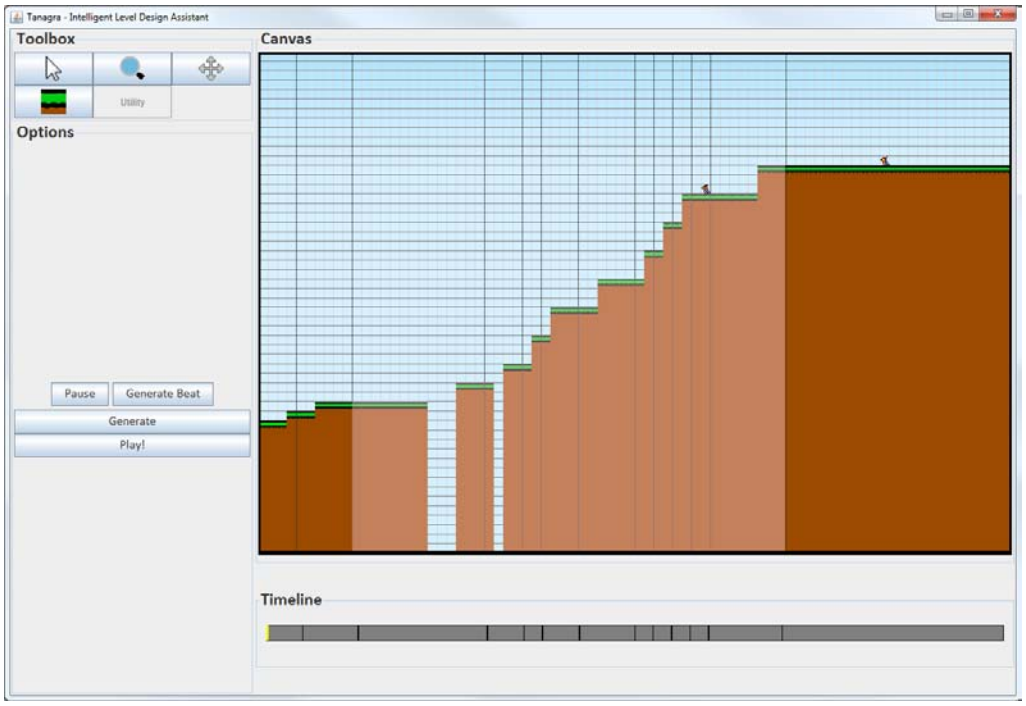

Figure 40. **Tanagra use case: rhythm changes (example 2).** Further rhythm changes are made by deleting beats at the center of the level.

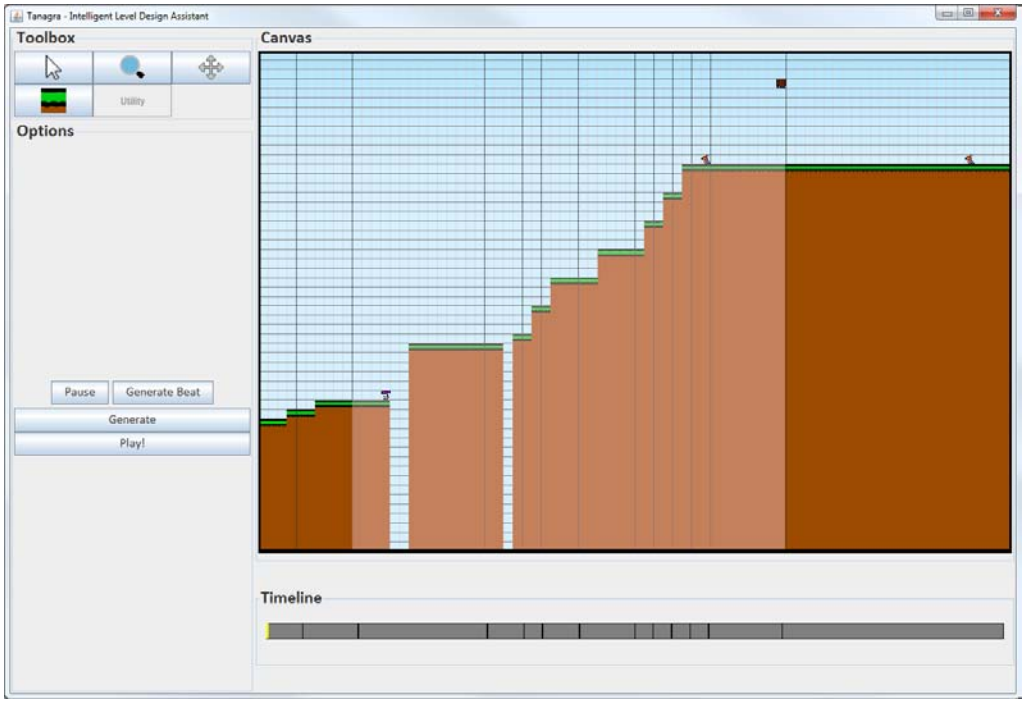

Figure 41. **Tanagra use case: selective geometry generation.** Finally, the designer regenerates geometry for only the third and second to last beat.

The designer then continues to experiment with different pacing by deleting some of the middle beats, the results of which are shown in Figure 40. Finally, Tanagra is capable of re-generating geometry for specific beats rather than the entire level. The designer completes the level in Figure 41 by regenerating geometry for the two beats she is unsatisfied with: the third and second to last beats.

Without Tanagra to assist her, the designer would have had to manually add and remove tiles corresponding to every piece of geometry. Prototyping pacing changes would be particularly arduous, as Tanagra provides support for altering pacing independently from level geometry; manual pacing changes involve a great deal of trial and error with different configurations of level components at different sizes. Also, Tanagra's ability to guarantee level playability means that at no point during her design process did the designer need to playtest the level to check that it was valid; all of her playtesting effort could be refocused on whether or not the level she was creating offers the experience she intended.

9 Interface Changes

In addition to the version of Tanagra described here, there are also several prototypes of the tool that have different user interfaces. The underlying foundation of Tanagra—the reactive grammars enabled by the use of ABL and Choco—has gone relatively unchanged across these prototypes, allowing us to experiment with a variety of different UIs. For example, we briefly explored several potential interfaces that would allow the direct modification and addition of constraints to the level, intended for more advanced users of the tool. However, these prototypes were largely unsuccessful, due to the difficulty of clearly and efficiently presenting the sheer number of constraints on a level to a user such that they could

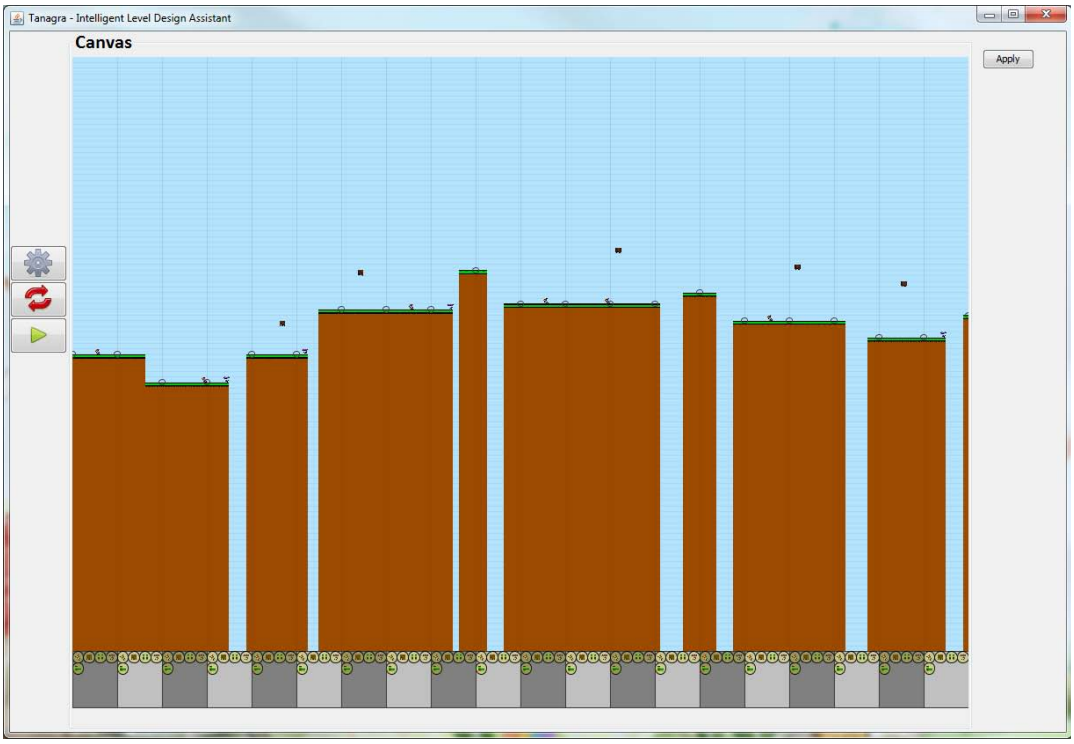

Figure 42. **A screenshot showing an updated interface for Tanagra.** Geometry pattern preference toggles decorate the beat timeline, which is now shaded differently every other beat. Anchor points at beat boundaries can be dragged up and down to manipulate platform positions. The left side of the screen has been replaced with a simpler interface with three buttons: generate, reset, and play.

understand them. We do still believe that this kind of interaction with the system is potentially useful; further research is required to find an appropriate interface to support it.

One newer prototype of the tool that we consider successful, discussed in this section, was created based on informal observations of Tanagra users. The first action most users would take was to click the generate button to get different ideas. Very few wished to draw in geometry themselves then auto-fill the rest of the level. We also saw a desire for increased control over the patterns that were chosen. This newer Tanagra prototype now has an interaction model of solely being able to interact with generated content rather than

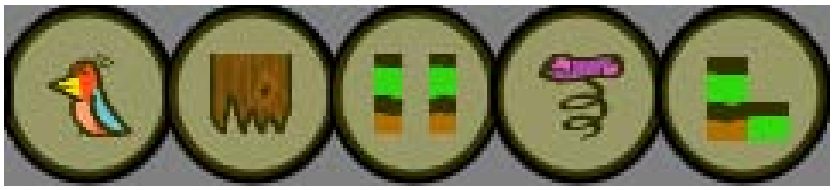

Figure 43. **Geometry preference toggles.** Toggles are in place for each geometry pattern: for enemies, stompers, gaps, springs, and falls. Yellow denotes a neutral preference, green denotes a preference, and red denotes undesirability.

drawing in geometry, and a geometry pattern preference system. A screenshot showing the new Tanagra interface is in Figure 42.

Starting from a generated level rather than a blank level canvas encourages the user to immediately experiment with editing a level, and provides an answer to the question of what the user should do first. The new Tanagra prototype begins by showing the user a generated level and inviting them to move geometry around and alter the level pacing by dragging anchor points that define the physical position of a beat boundary. This behavior replaces the platform dragging interaction in the earlier version of Tanagra. However, disallowing a user from drawing in their own geometry also removes some sense of control over the final result. Understanding the tradeoffs in designing appropriate interfaces for a mixed-initiative design tool remains an open research problem.

The geometry pattern selection area allows a designer to state their preferences for the kind of geometry that should appear in the level. There are five toggle buttons for each beat in the timeline, each of which corresponds to a particular pattern of geometry[*]. When the designer clicks on one of these toggles it cycles between its three available states: preferred, neutral, and unpreferred. These toggles are respected by the generator when the player hits

[*] Note that there is one extra geometry pattern in this version of Tanagra—the "fall", akin to Launchpad's geometry type of the same name—and multi-beat patterns are not available

the “apply” button to the right of the pattern selection area, although as they are only preferences, the generator might pick a different geometry pattern than the preferred pattern if it is needed to make the level playable. A zoomed in view on these toggles is shown in Figure 43.

Our experiences creating new interfaces for Tanagra highlights the importance of performing research into what are appropriate methods for interacting with a mixed-initiative system. The technology in Tanagra can be applied to a number of different potential interaction models; future research in mixed-initiative design tools must explore not only how to design the AI systems that can support the tools but also the ways in which human designers with to interact with them. I expect that these interaction methods will vary drastically based on the skill and experience of the intended user; Tanagra’s technology has the potential to support both novice and advanced users, but these groups would require very different ways of interacting with the technology.

10 Reactive Grammars

Stepping back from Tanagra, it is possible to generalize some of the lessons learned about creating reactive grammars using ABL. ABL’s hierarchical nature makes it simple to specify traditional grammars. For example, consider the following simple example:

```
A    →    x B y
B    →    B B z
B    →    x y z
```

This grammar can be expressed using only three main behaviors, one for each production rule, plus an additional supporting rule to print the terminals:

```
sequential behavior A() {
    subgoal print("x");
    subgoal B();
    subgoal print("y");
}

sequential behavior B() {
    subgoal B();
    subgoal B();
    subgoal print("z");
}

sequential behavior B() {
    subgoal print("x");
    subgoal print("y");
    subgoal print("z");
}

sequential behavior print(String str) {
    mental_act {
        System.out.print(str);
    }
}
```

Unlike in traditional grammars, it is also simple to specify sophisticated preconditions for each of the production rules; for example, it would be possible to limit the number of times that the first `B()` behavior is called by maintaining a counter and checking that counter in the precondition:

```
int numCalls = 0;
sequential behavior B() {
    precondition {
        (numCalls < 10)
    }
    mental_act {
        numCalls++;
    }
    subgoal B();
    subgoal B();
    subgoal print("z");
}
```

However, this is not enough in a reactive system, where the string might be modified during generation. Tanagra uses a manager to monitor for beats that have not been filled with geometry; this is akin to monitoring for non-terminals that have not yet been expanded.

This approach is made simpler by representing all of the non-terminals as WMEs, so that they can be easily checked by different success_tests and preconditions.

```
sequential behavior NonTerminalManager() {
    with (success_test {
              nonTerminal = (NonTerminalWME expanding==false)
          }) wait;
    subgoal expand(nonTerminal);
}

sequential behavior expand(NonTerminalWME n) {
    precondition {
        (n.getValue() == "A")
    }
    subgoal A();
}

sequential behavior expand(NonTerminalWME n) {
    precondition {
        (n.getValue() == "B")
    }
    subgoal B();
}
```

When non-terminals are replaced with terminals in their associated behaviors, the behaviors must now also either remove the non-terminals from working memory or flag them as expanded so that the `NonTerminalManager` does not repeatedly pick them up for expansion.

The most complicated aspect of the reactive grammar used in Tanagra comes where existing geometry has been placed by a human and must be consumed by the grammar and used in place of generated geometry rather than overwritten or ignored. The design idiom used here is to have at least two behaviors per non-terminal—one which generates a non-terminal, and one which consumes a non-terminal that is already present. The consuming behavior should have a higher specificity, to guarantee that it will be attempted (if

applicable) at a higher priority than generation. There may be more, depending on other conditions for consuming non-terminals:

```
sequential behavior print(String str) {
    specificity 2;
    precondition {
        (TerminalWME value==str)
    }
    succeed_step;
}

sequential behavior print(String str) {
    mental_act {
        TerminalWME t = new TerminalWME(str);
        BehavingEntity.getBehavingEntity().addWME(t);
    }
}
```

Note that this idiom for consumption of terminals works well in Tanagra because it has a predictable and well-structured output; the only non-terminals being monitored for expansion are beats, and it is known that there is a limited amount of geometry that can be assigned per beat, and when terminals in the grammar (i.e. platforms) are created by the designer they are assigned to a beat. This means that non-terminals are all assigned to particular terminals, which is not typical in traditional grammars. The final role of ABL as a reactive grammar in Tanagra—to "undo" geometry generation and assist in a search for a valid configuration of a level in the case of a need to backtrack—is also dependent on this kind of structure. How to adapt these strategies to a more general grammar requires additional investigation.

11 Future Directions

This chapter has described Tanagra, a new kind of mixed-initiative design tool operating in the domain of 2D platforming games. There are many exciting avenues for future research

in this area. A major issue that must be addressed in future work is that of how to handle conflicting user constraints; for example, if the designer carefully crafts two different sections of a level that has no valid connecting geometry, Tanagra currently merely informs the designer there is no solution. However, a better solution may be to suggest levels that incorporate each section in different locations, or relax constraints on one part of the level but not on another part. This is a challenging problem given Tanagra's architecture; numerical constraint solvers typically cannot state *why* a It can also be difficult to determine the "correct" action to take in certain situations; for example, when splitting a beat in half, how should any existing geometry for that beat be divided up? Important future work in the creation of tools like Tanagra is providing an ability to quickly view and choose between different design variants, especially in cases where there is no clearly correct response to a user-initiated change.

Related to this, there is plenty of further work to be done in improving the expressivity of Tanagra and ways to measure it. The current expressive range (see Chapter 7) of Tanagra is fairly large with only a few different geometry patterns; there is room to expand Tanagra by adding new geometry patterns that correspond to different player actions. For example, the canonical loop-de-loops from *Sonic the Hedgehog 2* (Sonic Team 1992), gaps that require double-jumping, and obstacles that involve wall-jumping would be interesting components to add and could greatly expand the number and variety of levels that can be produced. Adding these patterns would require creating new ABL behaviors for pattern placement and responses to geometry, and associated constraints that reflect a more sophisticated physics model. It would also require the support art assets and a formalization of the constraints

placed on the level by available art. For example, a tiled, sloped platform's width will be constrained by the slope of the art assets available; or a *Sonic*-style loop-de-loop can only be the size of its associated art asset.

There are also limits to how expressive the current implementation of patterns is—they currently support only static level components and NPCs with very simple behaviors, such as enemies patrolling along a platform. More complex, dynamic behavior would likely require additional methods for ensuring level playability, as it is hard to express such behavior as a set of numerical constraints. One potential way to address this is through the use of simulated players, running through the level using different initial configurations of components and making different choices throughout the level.

It would also be interesting to explore ways for designers to specify their own geometry patterns and constraints without needing to edit ABL or Choco code directly. Dormans supports users creating their own grammar production rules in his mission and space generator for action games (Dormans 2010) using a visual representation of the grammar; in a reactive grammar such as the one used in Tanagra, this representation would likely be more complex.

In summary, Tanagra is a mixed-initiative level design tool for 2D platformers that supports a human designer through procedurally generating new content on demand, verifying the playability of levels, and allowing the designer to edit the pacing of the level without needing to manipulate geometry. Important motivations in the design of Tanagra were the desires to focus on how the mechanics of the game relate to designing levels, to force

designers to consider aspects of player experience during design, and to encourage the easy prototyping and testing of different level designs by having the human designer interact with a PCG system. An expressive range evaluation of Tanagra is described in Chapter 7, and further plans for future work in creating design tools are discussed in Chapter 8, Section 2.

CHAPTER 6

ENDLESS WEB: PCG-BASED GAME DESIGN*

Chapter 1 discussed the possibility of a procedural content generator being used as an on-demand game designer, capable of making the game world deeply responsive to a particular player, where the world being explored is contextualized by the choices the player has made and presents challenges based on how the game has been played. This use of PCG is potentially powerful for everything from entertainment to education.

This chapter describes the process and result of building a game—named *Endless Web*—that uses PCG in this way, in which the PCG system is deeply tied into every aspect of the game's design. *Endless Web* was in part created as an evaluation of Launchpad as a level generation tool, to better understand its expressiveness. Forcing the system to operate within the context of a specific, polished game removes any abstractions that we could rely upon when originally creating Launchpad and permits an examination of its strengths and weaknesses as a level generator.

1 INTRODUCTION

What is a game that could not exist without procedural content generation, in the same way that *Crayon Physics Deluxe* (Purho 2009) and *World of Goo* (Gabler & Carmel 2008) could

* The description of *Rathenn* given in this chapter was previously published as a PCG 2011 workshop paper (Smith et al. 2011c); the description of *Endless Web* and design challenges arising from its creation are adapted and extended from an FDG 2012 conference paper (Smith et al. 2012b); the discussion of AI-based game design and its challenges first appeared in a 2011 UCSC Tech Report (Eladhari et al. 2011).

not exist without a sophisticated physics model? Such a game is referred to as a **PCG-based game**. Rather than using PCG to create fresh levels for fixed gameplay systems, one can seek to make the PCG system itself a focus of gameplay, reacting rapidly to player behavior—something the player comes to understand and manipulate through play. Using Hunicke et al.'s Mechanics, Dynamics, and Aesthetics (MDA) framework to provide a vocabulary (Hunicke et al. 2004), a PCG-based game is defined as follows:

> **A PCG-based game is one in which the underlying PCG system is so inextricably tied to the mechanics of the game, and has so greatly influenced the aesthetics of the game, that the dynamics – player strategies and emergent behavior – revolve around it.**

Creating a PCG-based game requires the careful co-design of both game and PCG system, as each has affordances, limitations, and requirements that influence the design of the other. For example, any bias in content created by the generator must be worked into the design of the game. Similarly, the game's design is likely to push additional requirements on the content that the generator should be able to create. This design process is called **PCG-based game design**, a form of AI-based game design (Eladhari et al. 2011), in which the capabilities of an AI inform the design of a game, and the game's design in turn provides further requirements for the AI system. In the case of *Endless Web*, the PCG system underlying the game design is the Launchpad level generator, described in Chapter 4.

Most games permit designers and art teams to have complete, fine-grained control over the structure and appearance of their content (e.g., the progression of levels and encounters

within them). The systems that produce emergence within games, such as interacting with combat mechanics or storytelling choices, are introduced and exercised through carefully-constructed content combinations. Building a PCG-based game instead involves relinquishing direct control over content and treating content generation as a game mechanic. Where traditional game design involves building spaces for players to play within, PCG-based game design involves creating the designer of that space while simultaneously determining the ways in which the player can interact with the space.

Creating a PCG-based game introduces a number of new design problems, for both the game and the PCG system itself. The PCG system is responsible for designing entire ranges of content that can be meaningfully controlled by both designers and players. Game design issues, from the moment-to-moment pacing of a level to difficulty curves, can no longer be solved through the direct manipulation of game content. Everyone involved in the creation of the game is impacted by the uncertainty over what content will appear to players: engineers cannot have special-cased code for particular regions of a level, artists and musicians must build modular assets that can be recombined at runtime. All aspects of the game are dependent on the algorithms used to create content, and designers become responsible for *directing* the content generator, in concert with other game systems, to create a desirable play experience.

A number of design challenges arose while creating *Endless Web* from the use of PCG as the core mechanic, due to this loss of direct control over the player's experience. The primary challenge came in determining how to balance control over the generator between the player and the designer, so that the player makes meaningful decisions but all of those

decisions lead to an engaging, directed experience. Other design problems include teaching the player to explore a generative space, providing well-placed goals to encourage this exploration, and art and audio issues that arose from having no knowledge at design time of how levels would be structured during play.

This chapter addresses the PCG-based design process in the context of designing *Endless Web*, describing the changes that had to be made to Launchpad so that it could be used in the game and the challenges encountered while designing a game around it. I also further define the difference between PCG-based games and games that use PCG to support more traditional designs.

2 Endless Web

Endless Web is currently the only game to my knowledge that involves the player directly controlling a procedural content generator during the course of play. The game is a 2D platforming game in which the player explores a literally infinite world; a generated environment is presented in front of the player no matter which direction they travel. Exploration is presented to the player as physical exploration: jumping onto springs, falling through tubes, or ascending in beams of light to discover new areas of the world. If the player continues moving to the left or right, an infinite world of gameplay unfolds in front of them. However, players are simultaneously exploring Launchpad's *generative* space, i.e. the many different kinds of levels that Launchpad is capable of producing by tuning different input parameters. Each time the player interacts with a special, glowing "tuning portal" the game propels her into a new part of the world that has been generated for her based on the choices she has made so far and how far she has progressed through the game. This form of

Figure 44. **The six different kinds of tuning portals**. Left to right: *enemies/conflict, platform hazards/betrayal, springs/entering the unknown, gaps/failure, moving platforms/losing control*, and *stompers/stress*.

exploration is enabled by Launchpad's parameterized control model that permits both experiential and compositional control of generated level segments (see Chapter 2, Section 1.1).

In *Endless Web,* the player takes the role of a member of a fictional race called Eidolons. Eidolons inhabit humanity's collective Dream; when humans fall asleep and dream, we wake up in their world. Humans and Eidolons share a symbiotic relationship, as nightmares disrupt the fabric of the Eidolons' world — they must seek out and wake up people who are trapped in their nightmares, thus releasing dreamers from their fears.

There are six different color-coded tuning portals in the world that correspond to each different challenge type (Figure 44); these tuning portals are each linked to one of Launchpad's parameters for compositional control. Each challenge type represents a different fear or negative emotion: *enemies/conflict, platform hazards/betrayal, springs/entering the unknown, gaps/failure, moving platforms/losing control*, and *stompers/stress*. Each fear has three different **tiers** of difficulty, which the player will encounter in order as they continue exploring deeper into a particular fear, and there are spaces between the tiers that slowly increase the probability for higher tier challenges to

	Tier 1	Tier 2	Tier 3
Enemies *Conflict*	Enemy moving along platform	Enemy jumping along platform	Enemy throwing projectiles
Platform Hazards *Betrayal*	Slippery platform	Decaying platform	Decaying, slippery platform
Springs *Entering the Unknown*	Spring	Moving spring	Moving spring nested in spikes
Gaps *Failure*	Narrow gap	Medium gap	Wide gap
Moving Platforms *Losing Control*	Moving platform	Moving platform, obstacle along path	Electrified moving platform
Stompers *Stress*	Single stomper with regular pace	Single stomper with variable pace	Cluster of stompers with variable paces

Table 3. **Difficulty tiers in *Endless Web*.** Each compositional parameter in Launchpad is linked to a challenge type and an associated fear. There are three different difficulty tiers for each type of challenge. Each challenge is color-coded: enemies are red, platform hazards are orange, springs are blue, gaps are green, moving platforms are yellow, and stompers are purple.

Figure 45. **Three different configurations of the Dream**. The leftmost screenshot shows two stompers and an enemy patrolling the long, unbroken platform. The middle shows a world with a lot of springs from all the tiers of difficulty. The rightmost has platform hazards and gaps. The background color of the world shifts colors to reflect the predominant challenges and difficulty tiers.

appear. For example, the *enemies* challenge type can be realized (in increasing order of difficulty) as a moving enemy on a platform, an enemy who jumps out at the player, and an enemy that hurls projectiles. Table 3 describes each tier for each challenge type and shows the color-coding of challenge types. The tiered difficulty system was added to Launchpad during the design of *Endless Web* to provide more variation in generated levels and more direct control over level difficulty (see Section 5.2.1). Figure 45 shows three drastically different configurations of the world based on the choices the player has made.

The goal of the game is for the player to rescue six dreamers who are trapped in their nightmares by exploring the Dream to find them. The six different nightmares are: *Body Horror, Broken Hearts, Creepy Crawlies, Dolls and Roses, Faceless Crowds,* and *Time and Death.* Each dreamer is hidden in their nightmare at a combination of tiers for three different fears; for example, the dreamer having a nightmare about creepy crawlies is

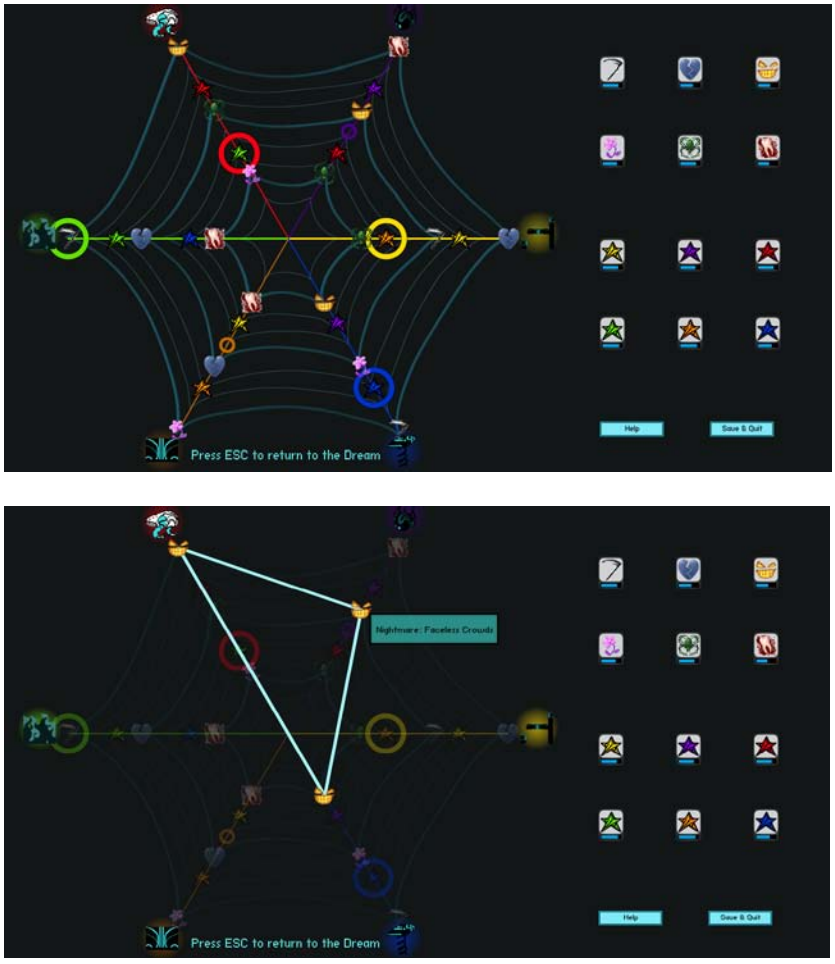

Figure 46. **The Web visualization.** A visualization of all the different goals available to the player that serves as a map of the generative space; thick lines denote tier boundaries, thin lines denote the spaces between tiers. The player has six primary goals to collect the dreamers (upper left) and six secondary goals to collect the powerups (lower left). The colored circles on each strand of the Web signifies the player's current location. When the player mouses-over an icon, all identical icons are highlighted.

	Moving Platforms *Losing Control*	**Stompers** *Stress*	**Enemies** *Conflict*	**Gaps** *Failure*	**Platform Hazards** *Betrayal*	**Springs** *The Unknown*
Body Horror		3		1	1	
Broken Hearts	3			2	2	
Creepy Crawlies	1	1	2			
Dolls and Roses			1		3	2
Faceless Crowds		2	3			1
Time and Death	2			3		3
Shield		1-1	2-1			
Place Block	1-1				2-1	
Float				1-1		2-1
Double Jump			1-1	2-1		
Time Slow	2-1				1-1	
Dash		2.1				1.1

Table 4. **The distribution of dreamer and power-up placement across Launchpad's generative space.** Numbers refer to the different challenge tiers. Power-ups are placed between main tier levels. Cells are color-coded by the color associated with each challenge type; darker colors indicate a more difficult version of the challenge. Numbers following dashes denote the number of spaces above a tier; these correspond to narrow lines in the web visualization of the generative space.

trapped at the intersection of the fears of losing control, stress, and conflict.

When the player reaches that intersection, the game presents her with a human-authored level representative of an individual dream that she must traverse to find the dreamer at the end of it. The six levels associated with the dreamers are the only levels in the game authored by a human as opposed to the level generator—these levels contain combinations of geometry that the generator would not be capable of constructing itself.

In addition to the primary goals of finding all the dreamers, there are six powerups scattered throughout the generative space as secondary goals at sections between the tiers. The powerups are hidden at a combination of tiers for only two different fears, so they are easier to reach while exploring to find the dreamers. Collecting a powerup unlocks a special ability that eases level exploration. Each powerup is related to a challenge type:

enemies/shield, platform hazards/place block, springs/float, gaps/double jump, moving platforms/time slow, and *stompers/dash*. Because of the relationship between challenges and powers, each powerup is placed just above the second tier of its associated challenge and just above the first tier of a different challenge. Table 4 shows the distribution of both dreamers and power-ups across the generative space. Figure 46 shows the visual representation of this generative space both the primary and secondary goals as icons at the intersection of web tiers with challenge axes; this visualization is presented to the player as a map.

The player's progress towards the different dreamers is reflected in both the art and music. Each level component has seven different art representations: one for each of the dreamers, and one for the undisrupted world (Figure 47 shows the seven different art assets for normal platforms and for springs). Tile art is procedurally selected at play time based on the player's proximity to the different dreamers. As the player gets closer to dreamers, the visual aesthetic coalesces towards that particular dreamer; for example, when the player is only a short distance away from the *Time and Death* nightmare, the level is likely to look like the screenshot in Figure 48a, but when the player is a far distance from all the dreamers, the art is far more jumbled (Figure 48b). Once a dreamer has been rescued, the art associated with that dreamer is no longer shown and is replaced with the art for the undisrupted world.

The music played is also procedurally selected; each time the player makes a choice, the game selects a track to overlay in the game based on that choice. Each tuning portal type has a track associated with it, and the volume of that overlay track is increased each time

Figure 47. **Themed art assets for *Endless Web.*** Top: The seven different art assets for non-hazardous platforms. The top asset is for the undisrupted Eidolon world, the remainder correspond to the nightmares (top to bottom): *Body Horror, Broken Hearts, Creepy Crawlies, Dolls and Roses, Faceless Crowds*, and *Time and Death*. Bottom: The seven different art assets for normal springs, in the same nightmare order as the platform assets.

Figure 48. **Run-time art asset selection.** As the player approaches a dreamer, the art style slowly coalesces to that of the nightmare (left: near the *Time and Death* nightmare). When further away from any one dream, the art is more jumbled (right: a level showing art for *Body Horror*, *Broken Hearts*, and *Creepy Crawlies*).

the player strengthens the fear associated with the tuning portal (and decreases each time the player weakens the fear). This leads to an atmosphere that sounds more cacophonous and chaotic when the level's configuration is more challenging and jumbled.

2.1 PCG Influence on Story

Endless Web's setting–the surreal landscape of dreams–also grew from the incorporation of procedural content generation and the mechanic of players changing the level geometry during play. There was also a technical limitation that the entire created world could not be held in memory (see Section 6.6.1) so content is generated off-screen and deleted off-screen as well, meaning that if the player turns around they will see different level geometry than was there before. Dreams provided a good setting that could explain these issues, since the landscape in dreams frequently shifts in unexpected ways. Initial story ideas involved the player taking on the role of a human exploring his own dreams and conquering his fears through facing them by pushing the generator to the maximum amount of a particular challenge.

3 The Use of PCG in Game Design

Typically, PCG has been used to replace or augment a human designer, providing additional variety and replayability in games that use traditional mechanics. However, *Endless Web* shows that it is possible to go much further than this. Rather than using PCG to create fresh levels for fixed gameplay systems, one can seek to make the PCG system itself a focus of gameplay, reacting rapidly to player behavior–something that the player comes to understand and manipulate through play.

There are many ways that PCG has been used in game design in the past. Using the definition of PCG-based games as a lens, we can examine four major, related aspects of PCG use—replayability, adaptability, player control over content, and game mechanics—to better understand the potential for PCG-based games. Games that use PCG fall along a

spectrum from “non-PCG-based” to “PCG-based”, and games may be PCG-based in one of these four regards but not the others. *Endless Web* was designed to be PCG-based in every aspect.

3.1 Replayability

Games such as *Rogue* (Toy et al. 1980), *Diablo* (Blizzard North 1997), and *Civilization IV* (Firaxis Games 2005) all use PCG to provide players with new experiences each time they start a new game. In these games, while the PCG system forms an important part of the player’s experience, the majority of player strategies are not greatly influenced by the generator. In roguelike games, the mechanics of the game are far more influential over player strategies than the fact that the players will be enacting these mechanics in different generated worlds. In *Civilization* games the majority of player strategies also form around mechanics such as unit build order, resource allocation within cities, and balancing the desire to grow an economy with researching new technologies and going to war with other civilizations. *Civilization IV*, for example, comes with several human-authored maps that still provide a great deal of replayability through these other, non-PCG systems. Roguelike games and the *Civilization* games sit at the non-PCG-based end of the spectrum with regard to replayability because player strategies are far more dependent on the static rule systems defined by the game designer than on the environments created by the PCG system.

However, there are also games for which replayability is at the core of the play experience; without the PCG system, players would be forming entirely different strategies and the purpose of the game would change. For example, *Robot Unicorn Attack* ([adult swim games] 2010) and *Canabalt* (Saltsman 2009) are both 2D platforming games that use procedural

content generation to provide players with new environments on each play through. In these games, the avatar moves across the screen without being controlled by the player; the player can only tell the avatar when to jump (and, in the case of *Robot Unicorn Attack*, dash) to avoid obstacles. The avatar is constantly accelerating, leaving the player with less and less time to react to changes in the environment. The goal of these games is to stay alive for as long as possible and beat any previous high score (which is directly correlated to the length of the play experience). PCG is key to crafting the player's experience; if the game took place in static environment, the core experience would be entirely different – it would become a game about memorizing action timing with challenges more similar to those of *Rock Band* (Harmonix 2007) or *Bit.Trip Runner* (Gaijin Games 2011). PCG is key to crafting experiences in games such as these because they depend on the player being surprised by the content seen on each individual playthrough.

3.2 Adaptability

A related concept to replayability is that of adaptability: the use of PCG to adjust content in reaction to player actions or skill levels. Adaptability is a way to improve the replayability of a game, and to diversify a game's player base by accommodating players who have a variety of play styles and skills. Much of the work thus far in creating adaptable games is done for the purpose of dynamic difficulty adjustment, and most of this takes the form of changing certain parameters to otherwise static content. For example, changing the amount of damage that a bullet does or the number of hits it takes to kill an enemy (Hunicke 2005; Kazemi 2008). However, there has been recent work in creating games that adapt to players through the use of PCG.

Left 4 Dead (Turtle Rock Studios 2008) incorporates an "AI director", capable of changing the spawn points and numbers of enemies in levels and also making some minor changes to environments, such as locking and unlocking doors, based on the game's interpretation of the player's experience compared to a desired experience curve (Booth 2009). The director is responsible for ensuring that the player is always engaging in challenges at the right time, thus maintaining the desired pacing and mood of the game. The use of PCG is fairly limited, however; the director can only alter certain aspects of a human-authored world, rather than create the world from scratch.

Galactic Arms Race (Hastings et al. 2009), described in Chapter 2 Section 1.1.2 as incorporating a generator that supports indirect control, is an example of a game that is PCG-based with respect to adaptability. This is a space shooting game in which the weapons evolve based on the player's inferred preference for certain weapon types. If the player uses a particular weapon more than others, newly generated weapons that the player picks up will be based on it. This leads to interesting weapons that correspond to different player strategies; for example, players who take a more defensive strategy are likely to see weapons that correspond well to that strategy. *Galactic Arms Race* allows players to replay the game many different times with different strategies that are supported by the generated content.

3.3 Player Control

Many games that incorporate PCG tend to have the generator behind the scenes, out of reach of players. *Rogue* and *Civilization* are both examples of this: while the player interacts with the generated content, there is no mechanism within the game for interacting with the

generator itself. This property is also true of some games with runtime PCG: *Minecraft* (Persson 2011) players have control over how they can explore the procedurally generated space, but no control over what they find there.

Games that use PCG for adaptation provide the player with an indirect form of control over the generator. For example, the *Polymorph* (Jennings-Teats et al. 2010) project, also discussed in Chapter 4, Section 5, adapts generated content based on its inferred understanding of player performance, so that players who are performing well are given harder content and players who are struggling are given easier content. While a player could choose to take certain actions, such as falling down gaps or being killed by enemies, that result in a level adapting to become easier, this is not the purpose of the PCG in this game. *Warning Forever* (Ohkubo 2003) provides another interesting example of indirect player control: each new boss is generated in a way that adapts to the previous means of destruction. While players cannot directly control the shape of the enemies they fight, they can adopt different strategies that result in drastically different enemy types.

Galactic Arms Race provides an interesting example of mixed control over the generator. While the weapons players can pick up in the game are procedurally generated, players also have access to the Weapons Lab, which lets them modify parameters for the weapons to customize them further. These customized weapons then become a foundation for newly evolved weapons (Hastings & Stanley 2010). In *Galactic Arms Race,* player control over the generator is indirect; the PCG is intended to be entirely invisible to the player, adapting to player strategies. In contrast, *Endless Web* offers direct control over the generator; it is intended to be something the player forms strategies around.

3.4 Game Mechanics

We can draw a distinction between PCG systems that augment traditional mechanics and those which enable new mechanics entirely. Consider, for example, the first-person shooter *Borderlands* (Gearbox Software & Feral Interactive 2009), which uses PCG to automatically generate a staggering number of weapons (approximately 17,750,000 unique combinations (Robinson 2009)) by combining different weapon properties. In this game the procedural weapon generation takes place within the context of familiar first-person shooter mechanics. *Borderlands* is not a PCG-based game in this regard.

On the other hand, Jason Rohrer's *Inside a Star-Filled Sky* (Rohrer 2011) is a good example of a game whose mechanics are PCG-based. Players navigate a space in which they can zoom into or out of recursively nested levels, each one generated from a seed passed from an object in a higher or lower level. Play events that take place on one level of play affect levels above and below themselves, which in turn reinforce or reseed the procedurally generated levels that the game is building. It would be impossible to hand-author this game, as this mechanic of the game is integrally tied to the PCG system.

3.5 *Endless Web* as a PCG-Based Game

Recalling the definition of a PCG-based game given earlier in this section, understanding *Endless Web* as a PCG-based game requires an analysis of it in terms of its mechanics, dynamics, and aesthetics. Mechanics-wise, *Endless Web* is consciously positioned within the traditions of the 2D platformer. The basic actions available to the player are those typical of existing platformers, including jumping over gaps, killing enemies, and avoiding stompers. The level elements used in the game are also commonly seen in traditional platformers;

enemies patrol back and forth along platforms, stompers descend from the "ceiling" of the level, and springs propel the player to greater heights. However, the major choices that player make in *Endless Web* are relatively unrelated to these platforming mechanics. *Endless Web* is fundamentally a game about manipulating a generative space, and the core mechanic involves the player deliberately choosing to influence the generator in different directions through interacting with the glowing tuning portals. This core mechanic is intertwined with the generator; each tuning portal the player interacts with directly changes parameters to the generator.

A key aesthetic in *Endless Web* is a sense of exploration and wonder, both in terms of physical exploration and uncovering the generative space of the game. As the player progresses through the game, the generator provides increasingly challenging combinations of content, and the variety of potential content is quite high. The world the player is exploring is endless; the player could choose to keep moving in one direction forever and the generator would continue to provide content appropriate to the choices the player has made. This aesthetic could not be achieved without the use of procedural content generation; there is a limit to the amount of human-authored content that can be made for a game and thus the extent to which players can explore a world, and the generator makes it possible to remove this limitation.

This aesthetic comes from the player's interactions with the generator using the tuning portals and desire to achieve the goals that are scattered throughout the generative space. The dynamics of the game involve the player building strategies around which direction to push the generator at any given time. To achieve each goal, the generator must be

configured to a particular location in generative space, but there are many ways for the player to reach these locations. The player can choose to manipulate the generator in different ways in order to achieve goals in different orders, or to reduce the difficulty of a particular challenge that isn't needed for the current goal. Thus, the player is building strategies around the procedural content generator to make sure that the content seen is of an appropriate challenge and interesting composition. The generator also provides different content on each playthrough, as in *roguelike* games – even if two different players make exactly the same choices in the same order, they will still have a different play experience because the generator can create many different variations of content for each combination of parameters. These two aspects of the game lead to *Endless Web* being highly replayable.

4 The PCG-Based Game Design Process

PCG-based game design is a specific instance of AI-based game design (Eladhari et al. 2011), where the PCG system is an artificial intelligence simulating the role of a game designer. When creating AI-based games, the design of the AI system needs to be incorporated into the game design process. This method of the iterative co-formation of AI and game design is called the AI-based game design process.

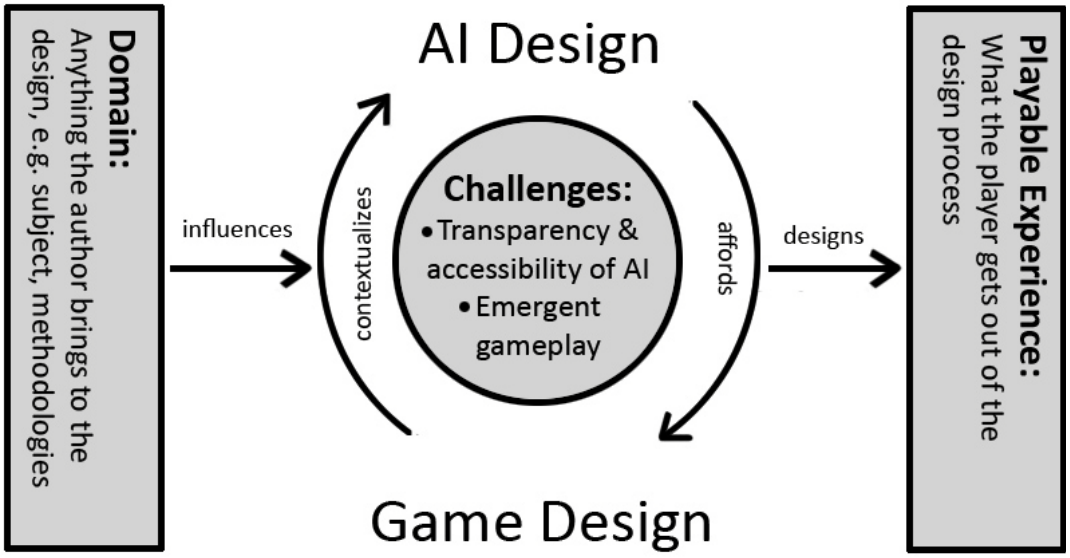

Figure 49. **The AI-based game design process.** The inputs to the process are domain knowledge relevant to both the game and the AI system, and the output is a playable experience. AI Design and Game Design are iterated on and inform each other throughout the process, especialy with a view to resolving two key challenges in AI-Based Game Design: Transparency of the AI and handling emergent gameplay.

Although AI systems can help us explore the expressive range of game design, it is impossible to merely write a new AI system and expect that this alone will create a new experience. AI systems require context to work within, providing motivation for them to act and data for them to interpret. In a game, this context is provided by the design of the game world in which the AI operates, subject to constraints chosen by the game designer. If the AI makes choices independently of the environment it is situated in, the actions will lack context or meaning, and will make the system feel autonomous (if overly predictable) or schizophrenic (if lacking an overall structure) (Eladhari et al. 2011). The design of an AI-based game needs to have an impact on the space within which the AI operates, and give context to the actions made by the AI. By allowing the game to inform the design of the AI,

the AI can be contextualized to maximize the amount of intentionality the player can read into the system. AI-based game design follows a process described in Figure 49.

4.1 The Influence of Domains

Three main types of domain knowledge have an impact on both the AI design and the design of game mechanics:

1) The subject matter of the game,
2) Genre-specific design conventions, and
3) Established AI techniques.

For example, a game basing its core gameplay on musical theory can open very different types of play activities than a game based on collaborative storytelling. Design conventions, such as typical challenges that face players in a particular kind of game, also shape what affordances designers create within the systems. The design of the AI system used in the game also influences the design of the game, either through further constraining or opening up the game design space.

Endless Web's subject matter is a surreal dream world where each challenge type corresponds to a particular fear, and dreamers are trapped at configurations of these fears that correspond to nightmares. This subject was initially chosen in large part due to the use of procedural content generation and a need to explain to the player why it is that the composition of the world is constantly shifting. The domain of a dream world led to a number of decisions regarding the art and music in the game, including the procedural selection of art and the procedural mixing of different music tracks. The design is also

heavily influenced by the genre conventions of 2D platformers. The level representation used in *Launchpad* is derived from the analysis of rhythm and level structure in popular 2D platformers, as described in Chapter 3.

4.2 Entering the Design Loop

At the core of the AI-based game design process is a loop involving the AI design and game design informing each other in an iterative manner. As it is a loop, there is not one defined entrance point: it is possible to enter this loop from either the game design or the AI design side. For example, *Mismanor* (Sullivan et al. 2011) is an AI-based game that was motivated on the design side by a desire for modeling more meaningful interactions in role-playing games and making quests as playable as combat.

Launchpad was a mature content generator when the design of *Endless Web* began. The goal in creating *Endless Web* has been to study how PCG can lead to new playable experiences and to learn more about the *Launchpad* level generator to better inform newer PCG-driven design tools. However, entering the loop from the AI system side will still result in changes being made to the generator as the game's design develops. As a result of creating *Endless Web*, a number of changes have been made to the *Launchpad* level generator to support iterating on a game's design. Without a rough design of the AI system, it is impossible to flesh out the mechanics of the game. Similarly, without an idea of how the game will work, it is impossible to know what should be modeled in the AI system.

4.3 Design Challenges

There are two major challenges that must be kept in mind throughout the creation of any AI-based game: having an appropriate level of transparency for the AI system, and addressing issues that arise from emergence. In this section, these challenges are discussed more abstractly; the way we addressed these issues in the creation of *Endless Web* are described in Section 4.3.

4.3.1 Transparency of the AI System

In *Expressive Processing*, Wardrip-Fruin discusses a problem called the *Tale-Spin effect*, where a player fails to read intentionality into a complex AI system. This occurs when there is no "means for interaction that would allow audiences to come to understand the more complex processes at work within the system" (Wardrip-Fruin 2009). Finding the appropriate level of transparency for the AI system, or the amount of the AI system that is exposed to the player, is a delicate balancing act. It is vital for players to understand that there is a deep AI system that understands their actions and allows them to make meaningful choices, otherwise their actions feel shallow and the responses from the system feel random. However, from a gameplay perspective, it is important not to overwhelm the player with too much information about the AI system.

4.3.2 Handling Emergent Behavior

There are two major qualities of emergence discussed here with regard to AI-based game design: the emergent gameplay evident in AI-based games, and using AI-based game design to understand emergent qualities of the AI systems themselves. Emergent gameplay refers to the complex situations in games that can emerge from the interaction of relatively simple

game mechanics. Often, in robust systems, players reach spaces permitted by the game mechanics but not specifically anticipated by the designer. Emergence within the AI system itself occurs when the system procedurally generates content with which the player will interact. Due to the emergent nature of AI-based games, it is important to design with this quality in mind.

5 Designing Endless Web

The design process for *Endless Web* followed the principles of AI-based game design, with Launchpad as the AI system. Launchpad's existing design offered two main design affordances: variables that could be exposed during gameplay, and a rhythm model that supports altering pacing. The design process began by examining these affordances; however, designing the game quickly uncovered further requirements for Launchpad. This section describes the design process used in creating Endless Web, and how the AI system informed the game design and vice versa.

5.1 Early Prototypes

There were two early prototypes created for *Endless Web* that explored the use of PCG as a mechanic: a technical prototype was created for understanding how manipulating different parameters influence the player's experience, and a playable prototype named *Rathenn* was made to answer questions about methods for the player to expore and understand Launchpad's generative space. The two prototypes also helped us understand how effective the generator was for creating sufficiently varied and challenging levels. These two prototypes greatly informed the changes made to the level generator and later design decisions while creating *Endless Web*.

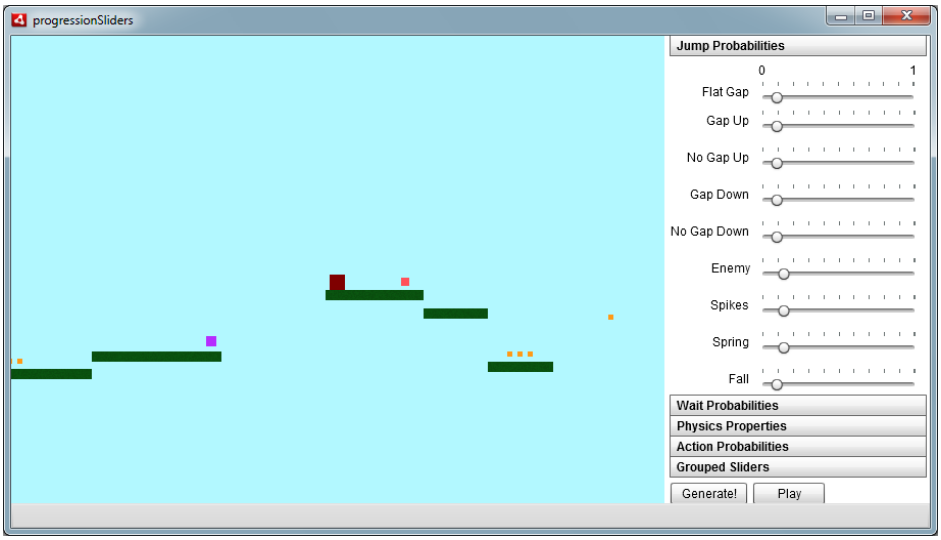

Figure 50. **Early *Endless Web* technical prototype.** The technical prototype built to understand the impact of altering parameters on the player's experience. Sliders on the right control every parameter of the generator. The area on the left allows the designer to play levels as soon as they are generated.

5.1.1 Technical Prototype

The technical prototype presented all of the available parameters as sliders on the right side of the screen to make it easy to rapidly play with different parameter configurations (Figure 50). This prototype was useful in determining which parameters were most obvious when changed, the relationships between parameters, and how to combine sets of parameters to make the generator less intimidating to players. For example, parameters to Launchpad include the frequency of wait and jump actions in a rhythm that are set separately from parameters for different geometry components. While a visualization of the generative space (see Chapter 7) has shown relationships between Launchpad's parameters through an analysis of thousands of generated levels, creating this prototype made it possible to understand the ramifications of those dependencies for the player's experience.

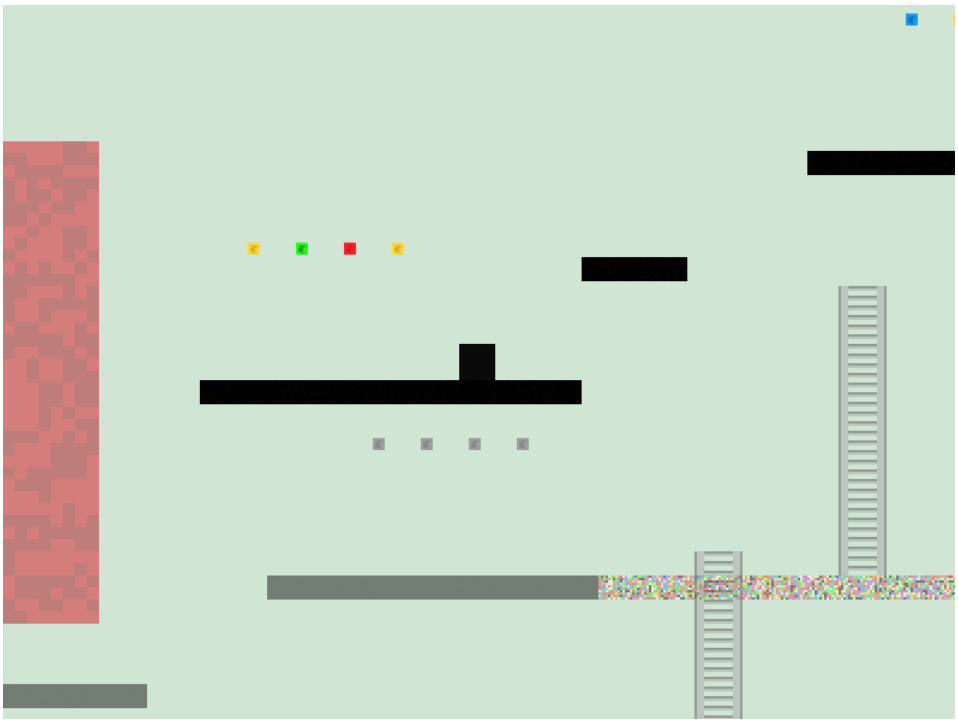

Figure 51. **A screenshot from *Rathenn*, the early playable prototype for *Endless Web*.** The darker colored level geometry is the current playable level; the lighter geometry is a path that the player could have taken but chose not to. The black rectangle in the center of the screen is the player, multi-colored boxes are coins, the large red rectangle on the left is a stomper, and there are ladders along a choice-platform visible in the backgrounded level.

5.1.2 Rathenn

Rathenn can be seen as the first iteration of *Endless Web*; it uses an unmodified version of Launchpad. As players move laterally through the level, they overcome a series of challenges in a procedurally generated level segment. At the end of each segment, the player encounters a set of ladders that she can choose from to climb (precursors to *Endless Web's* tuning portals). These ladders cause a new segment to be generated, with updated properties from the prior segment as determined by the color of the ladder. When the player reaches the end of a ladder, a new segment appears in the game for further

exploration. The background color of the world also changes as a result of taking a colored ladder. Figure 51 shows a screenshot of Rathenn during play.

There are six colors of ladder corresponding to three sets of parameters that can be altered. Note that these six ladder types are different from the types of tuning portals in *Endless Web*. The two colors belonging to each set are complementary colors, denoting opposite changes. The color of the ladders at the end of each segment is determined by how many coins of each color the player has collected. For example, if the player has more red coins than green coins, a red ladder will appear instead of a green ladder. There is always one ladder per decision type available to the player. Activating a ladder causes a shift in the color of the background towards the chosen color. Players are thus visually cued as to the impact of their decisions on system parameters. The ladder types are as follows:

- **Red** More enemies, fewer gaps
- **Green** More gaps, fewer enemies
- **Blue** More jump actions, fewer wait actions
- **Purple** Avatar can run faster and jump higher; springs are more likely to appear
- **Yellow** Avatar can run slower and not jump as high; springs are less likely to appear

Early response from playtesters for *Rathenn* was positive, showing that the core experience was compelling enough that it was worth pursuing the idea further. Players enjoyed the boundless exploration of space, and feeling that their decisions were changing the composition of the world. Thus, the core experience from *Rathenn* – mapping exploration of the generative space to navigation in physical space – remains in *Endless Web*. However,

much of the rest of the game is different based in large part on the lessons learned from this prototype. The major differences between the prototype and the game are described here, along with a brief description of how the problem was addressed. Each design challenge is described in more detail in the remainder of this section.

1) *Endless Web* has goals placed within the generative space to guide exploration. This was in direct response to playtesting results showing that undirected exploration was not compelling enough for players to continue.
2) *Rathenn* uses only the background color to show players where they are in generative space; however, this led to problems where the background color can be achieved in multiple ways. For example, the background color might be orange because the player has used exclusively orange ladders, or it might be because the player has used a combination of red and yellow ladders. While the shades of orange are different, the difference is far too subtle for players to perceive. *Endless Web* added both a map that visualizes the generative space and visual and audio "fingerprints" for each configuration of the generative space.
3) The kinds of ways players could influence the generator changed drastically from *Rathenn* to *Endless Web*, both in terms of the parameters that can be manipulated by the player and the mechanisms used for exploring the generative space. This was in part to provide a more organic-feeling exploration of the space, and to be able to retain control of certain aspects of the content within the game instead of giving control over everything to the players.

5.2 Modifications to Launchpad

Lessons from the *Rathenn* prototype also guided a number of changes to the Launchpad level generator so that it could support the kind of play experience desired in *Endless Web*. The primary change was the creation of a tiered difficulty system, which both increased the variety of content and provided an ability for the designers to better direct the difficulty curve of the game. There were also more minor changes to Launchpad's timing system and coin placement algorithms.

5.2.1 Tiered difficulty

Launchpad was originally designed to choose from all components according to a probability specified by the designer. The tiered difficulty system altered this by adding more components at different difficulty levels, and having the generator choose the difficulty of a component independently from the component type. *Endless Web* keeps track of player progress into a particular challenge and requests the appropriate tier of difficulty from Launchpad when the player uses a tuning portal. The player can choose whether the difficulty of that challenge will be increased or decreased before entering the portal by hitting a direction box (similar to a question mark box from *Super Mario World* (Nintendo EAD 1990)). Raising the difficulty of a challenge type raises the probability that a more difficult component will be chosen. This design decision also solved the problem of variety in levels by tripling the number of components available to the generator while maintaining uncertainty about which components will be chosen.

5.2.2 Relaxing timing precision

One of the major parameter dependencies uncovered in the expressive range evaluation in Chapter 7 is the relationship between beat timing and spring placement: if beats are placed too closely together, springs cannot be chosen for those beats because the avatar will still be in the air when the next beat occurs. Because spring occurrence was a parameter that we wanted players to have control over, this timing restriction was relaxed in the version of Launchpad used in *Endless Web.* The amount of time spent in the air for a particular challenge type is ignored when calculating the position of level components. This change does alter the exact rhythm felt by the player, but does not otherwise have a large impact on the overall pacing of a rhythm group. A rhythm group with 5 closely-spaced beats will have those beats slightly further apart if they are all realized as springs than if they are all realized as gaps. However, this tradeoff seems worthwhile given the extra amount of control the player can have over the appearance of different geometry types.

5.2.3 Coin placement

The final change made to Launchpad was a modification to its coin placement algorithms. In addition to placing coins at the peak of gaps and along long platforms (as in the original version of Launchpad), *Endless Web*'s Launchpad also places them along the path of moving platforms. This is done to guide the player through the world when the moving platforms might be initially off-screen and therefore invisible for a few seconds while they are moving on-screen.

6 Design Challenges

Building a game around a PCG system introduces some unique design challenges, including how to best integrate the particular PCG system into the game, how to design for always having desirable content while still giving the player control, and how to teach the player to understand entirely new rules within common genre conventions. Some problems arise from the abilities and biases of the generator, while others arise from technical decisions made to support an infinite world. This section discusses these challenges in the context of designing *Endless Web*.

6.1 Using PCG as a Mechanic

The primary design goal for *Endless Web* was to create a game that could not exist without procedural content generation. The design process began with an analysis of the capabilities of Launchpad, with a view to how these capabilities afford new game mechanics. Launchpad's key feature is how the input parameters are directly tied to the levels that it can create, and altering these parameters was identified as a core mechanic. Early design ideas involved altering the parameters indirectly; for example, one proposed idea was a platformer game for Facebook where beat density would be determined from the frequency of status updates and components would be selected based on the tone of posts. However, in keeping with the definition of a PCG-based game as one whose dynamics are strongly influenced by the procedural content generator, we decided instead to provide the player with relatively direct control over Launchpad's input parameters, thus offering players the opportunity to build strategies around the generator.

This also fit in with the desired aesthetics for the game. PCG is well-suited for an exploration game due to its capability for creating new territory for the player to explore infinitely in any direction. By giving players control over Launchpad's input parameters, they are able to explore not only the physical space of the world but also Launchpad's *generative space*. The technical prototype described in Section 5.1.1 was used to determine which of Launchpad's parameters should be exposed to the player. The parameters chosen were those that provided the greatest variation in player experience while minimizing potential undesirable dependencies (e.g. we do not allow players to modify the physics of the game despite being a noticeable change because it has too many implications on the availability of different level components).

6.2 Balancing Control for a Generative Space

Using PCG as a mechanic requires the player to have control over the kind of content that appears in the game. However, there is a measured amount of control that the designer should retain over the generator as well, to handle design concerns such as difficulty or pacing. In *Endless Web,* the game state determines the value of certain parameters to the generator while the player controls others through play. The player retains a great deal of control over the generator, but within the boundaries set by the game's design.

Launchpad's separation of pacing from level components seemed a natural fit for drawing a boundary between player and designer control. Thus, the player was provided with control over the components that appear in the level, rather than rhythm parameters. Control over the pacing of the level was kept in the game, as level pacing is a global consideration related to difficulty, which we wanted to be able to manipulate ourselves as game designers rather

than leave in the hands of the player. Any PCG-based game must have a similar separation in its underlying generator so that the designers still have control over some aspect of player experience.

6.2.1 Designer Control: Difficulty and Pacing

Retaining control over difficulty and pacing was a response to playtesting feedback from earlier iterations of the game. There was not enough variety in the levels being created, and there was no clear difficulty progression. This problem was addressed in two ways: through the creation of a tiered difficulty system in Launchpad and by providing the game with control over the level's pacing.

While variety and unpredictability in content is considered good, that variety must still be carefully structured to provide an engaging and fair experience with an appropriate level of challenge. *Endless Web* controls the rhythm and pacing parameters by slowly increasing them as the player reaches his goals. Level segments start out as short, slower paced segments and change (as dreamers are rescued) to be longer and faster paced. Note that increasing the parameters actually alters the *probability* that a segment will have the appropriate length and density, not the actual frequency itself. This adds some extra variation and can surprise the player with a more challenging than usual section early in the game, but on average the segments meet the intended difficulty curve.

6.2.2 Player Control: Level Structure

While the game retains control over the pacing of the generated levels, the player is given control over the overall composition of the world. Each time the player uses a tuning portal,

the probability of the appearance for the corresponding component is altered. Launchpad then generates 50 different candidate level segments and returns the one that most closely meets the parameters for component frequency. This provides players with appropriate feedback about the changes they are making to the generative space, resulting in players feeling like they are controlling what they see in the game.

6.3 Navigating a Generative Space

The aesthetic of exploration in *Endless Web* is designed to feel natural and organic. The tuning portals are designed to be familiar level elements that, while clearly marked as transitions to a new location, still feel like they belong in the game world. Using familiar level components provides a cue to players about what they should expect the generator to do when they are used. We found that one of the most challenging aspects of navigating generative space is the lack of waypoints or landmarks that can be used when navigating a physical space, since all of the content is generated. This issue was further addressed through the use of art assets and audio to provide each configuration of the generative space with a unique "fingerprint".

Important decisions that made generative space navigation less frustrating were made when designing the algorithm for how tuning portals should be scattered throughout the world. In *Endless Web,* to make exploration feel more organic and well-paced, there is only one tuning portal that appears between level segments, allowing the player to keep moving through the level to find more portals.

Originally, these portals were placed entirely randomly. However, the entirely random placement was jarring to players. One player described the experience as "being at the mercy of a random number generator". Players would frequently express frustration at being unable to find the one portal they needed to reach a goal. Tuning portal placement needed to be done *intelligently*, not randomly. Indeed, while *Endless Web* uses random numbers frequently, they are always part of a directed experience.

This problem was addressed by probabilistically placing tuning portals based on how likely the game judges the player is to need them. For example, if the player is only two portals away from reaching a goal, the game is more likely to show the player the portal that he needs, intentionally assisting the player in reaching the nearest goal. However, if the player ignores the portal and continues moving to the left or right, he will see the full sequence of portals before seeing any repeats. The set of available portals is reset whenever the player uses one. This guarantees that the player will always be able to find the portal he wants to use, but is more likely to see the portal that he needs.

6.3.1 Making the PCG Visible

As mentioned in Section 4.3.1, a key issue in building games around AI systems is avoiding the Tale-Spin effect; the underlying AI system must be transparent enough that the player can understand what it is doing. The Tale-Spin effect occurs when there is no "means for interaction that would allow audiences to come to understand the more complex processes at work within the system" (Wardrip-Fruin 2009, p.419).

The decision to make exploration completely seamless means that the player never sees Launchpad in action. There is no geometry popping into the screen, and no way for the player to see Launchpad construct levels and choose between different options. Thus, *Endless Web*'s only path to avoiding this effect is making sure the player's choices have a clear and predictable effect on the world. Playtests have shown that this works well at lower tiers, when elements are first introduced. However, it is currently unclear how well this works at higher difficulty tiers, or when all the tiers are at equal values. When this occurs, the probability of each component appearing is so similar to the others that it can be hard to see an immediate impact from choices. We intend to better understand how players interpret the consequences of their choices in future work.

6.3.2 Reusing Genre Conventions

Another challenge faced in *Endless Web* was the player's interpretation of certain genre conventions. Experienced platformer players naturally tend towards moving to the right instead of the left, and assume that falling down gaps leads to death. Since three of *Endless Web*'s tuning portals transport the player downwards, and the player always has the ability to move to either the left or right at the end of a portal, this presented problems in our design. Early versions of the tuning portals involved the player falling down gaps. This issue was addressed by changing the downward moving portals to have the player move through a tube instead of falling in mid-air, as traveling through tubes is a platformer design convention that implies player safety (e.g. the Chemical Plant Zones in *Sonic the Hedgehog 2* (Sonic Team 1992)). Playtests run before and after these changes show that using tubes for downward portals reduces player confusion. However, the issue with players always

wanting to move towards the right has not been resolved; our lesson from this experience has been to not fight *against* genre conventions but rather either work within them or invent new ways to communicate with the player.

6.3.3 Goals for Exploring Generative Space

The dreamers in *Endless Web* are scattered throughout the generative space. Goals are placed in generative space rather than physical space to reinforce the design goal that players should be exploring the capabilities of the generator, and that physical position in space is immaterial. The tiered difficulty system offered seemingly obvious goal placement; with six challenge types and six dreamers, there was initially one dreamer at the end of each challenge, and the player would be rewarded for finding a dreamer by giving her a powerup.

However, placing goals at the end of the web strands was not interesting enough. It provided the player with no motivation to actually *explore* the generative space because they were always simply aiming to max out each axis. This problem prompted us to move goals to be hidden within "layers of the Dream" at different world configurations, and de-coupled powerups from the dreamer locations. While this added more complexity to the game, requiring the player to learn another system , this did successfully encourage players to explore the generative space and see more interesting configurations of content. The powerups became secondary goals that helped the player through the game, although they are not required. Since Launchpad has no knowledge of the powerup system, there is never a level that cannot be completed without a powerup.

6.4 Teaching the Player

The hardest aspect of designing *Endless Web* was teaching the player how to understand an entirely new game genre that, on the surface, appears identical to other, more familiar, platforming games. After the main part of the game was designed, we introduced a tutorial series that guided the player through the different systems of the game. The first phase of the tutorial teaches the player the controls and core platforming concepts, and requires the player to take the platform hazards tuning portal. This section of the tutorial is hand-authored. While it could have been made with a different level generator that would sufficiently control the content and placement of instructions, we don't expect players to replay the tutorial level once they understand the mechanics so a procedurally generated tutorial would offer little value over a hand-authored one.

After the player uses the platform hazards tuning portal, the game continues with procedurally generated content. The Eidolon is given a small goal to find his elder tutor, who is waiting at a minor configuration of the world. The player cannot see the entire Web at this point, only a small subsection of it. The tutorial familiarizes the player with the concept of using tuning portals to change the world configuration and aligning the world to a specific configuration. After the player finds the Elder, the rest of the game opens up and the player is given the main goal of finding all the dreamers.

Designing the Web was particularly challenging, as players expected it to be a physical map of the world. Many early playtesters assumed that the positions on the web corresponded to physical positions in the world, and that by moving upwards in space they would move "up" on the Web. This is another example of genre standards confusing players, in this case

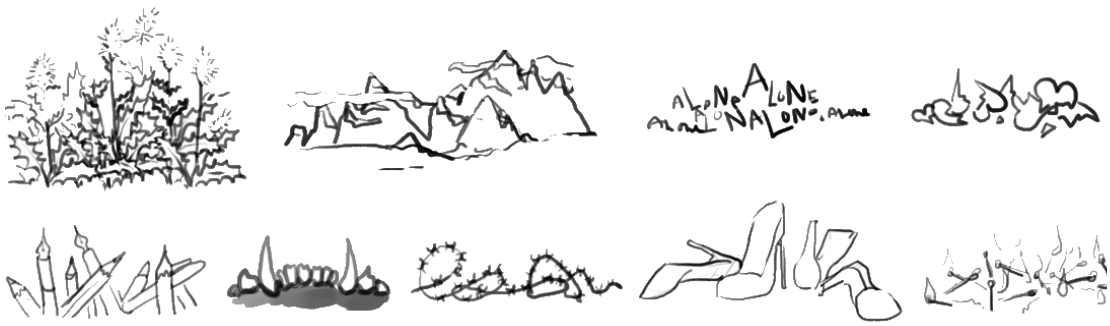

Figure 52. ***Endless Web*** **concept art.** A selection from the 54 original concept art sketches for the spikes challenge.

by repurposing physical direction to mean something completely different. It is not immediately clear to players that the world they are exploring is endless, as it looks identical to other platformers they have played in the past. To solve this problem, the Web was never referred to in the game or any tutorial text as a "map" and additional tutorial information was added to the Web screen itself.

6.5 Art Influence and Challenges

Having a visually compelling world was critical to our design goals; we wanted players to have the sense of being in a surreal dreamscape, surrounded by familiar nightmare motifs. Therefore there is human-created art incorporated into the game, rather than procedurally generated assets. These assets were designed to be modular so that they could be procedurally selected during play. Modular art design introduced a large challenge for the art team, as there is no way of knowing ahead of time the exact placement of platforms or enemies.

The art direction had a large influence on the final story of the game. One round of early concept art included 26 ideas for what spikes might look like – from broken hearts and spiky

high heels to teeth and electric fences (Figure 52). The diversity in art we saw for spikes influenced our decision to introduce the nightmares to the game, and the role of the player shifted from being a single human dreamer to a creature who is trying to find and rescue multiple dreamers trapped in these nightmares.

Designing new art assets also revealed implicit assumptions and inconsistencies in Launchpad's physics system. Launchpad was designed to support changing physics parameters (e.g. the player's movement speed or the sizes of level components) without altering the generator at all. However, the original design for Launchpad had not accounted for constraints among these parameters, and when new sizes of art assets were added, the physics system in Launchpad broke due to failing to enumerate these constraints. For example, the height of enemies must be set such that they are no taller than the player's maximum jump height, otherwise the player will not be able to jump over them or jump on top of them. Using art assets that had a different ratio from the originally intended size for Launchpad revealed such inconsistencies, and prompted a redesign of the physics system such that these constraints are taken into account.

6.6 Technical Design Decisions

In addition to the game design issues detailed above, there were also two key technical design decisions that guided the creation of *Endless Web*: memory limitations for an infinite world and the potential for delayed response from the generator.

6.6.1 Memory Limitations

With an infinite world, it is impossible to store all of the generated content in memory. Although a stored seed could have re-generated content on demand, simply deleting all off-screen content provided a much simpler architecture and reduced any need for worrying about memory management. As mentioned earlier, this architecture decision played a large role in initial story concept development. Endless Web's setting—the surreal landscape of dreams—also grew from the incorporation of procedural content generation and the limitation that content is generated off-screen and deleted off-screen, meaning that if the player turns around they will see different level geometry than was there before. Dreams provided a good setting that could explain these issues, since the landscape in dreams frequently shifts in unexpected ways.

6.6.2 Client/Server Split

While Launchpad is capable of producing levels extremely quickly—usually requiring less than a second—any lag in response from the generator must be handled in the game. This lag was compounded by the use of a client/server architecture; Launchpad is a web service that is called by *Endless Web* whenever a new level segment is needed. A new segment is required once every 5-20 seconds. Having Launchpad as a web service provides flexibility in updating the game and testing it with different PCG techniques, and also permits gathering of gameplay metrics. However, it does introduce a potentially significant problem in handling server lag and lost connections.

Auto-saving the game at every transition point, and requiring each tuning portal to be able to extend indefinitely while waiting for server response, addressed this problem. For

example, the *enemies* tuning portal involves the avatar being lifted up in a beam of light. This beam extends off-screen until the server has responded to the generation request. This maintains the sense of seamless exploration as long as possible. If the request fails, the player is placed in a human-authored level segment that tells the player they've lost their connection to the dream.

7 General Lessons

While the discussion regarding PCG-based game design thus far has largely been specific to *Endless Web*, there are three main lessons that can be drawn for the future creation of PCG-based games. The first two lessons have applicability to AI-based game design in general; recall from Section 4.3 that the two main challenges in the AI-based game design process involve designing for emergence and maintaining an appropriate level of transparency to the AI system.

Balancing control over content. Dan Kline, speaking of designing the game *Darkspore*, noted that all aspects of the player's experience should be *directed*—purely randomized aspects of content stand out to the player as undesirable (Kline & Hetu 2011). Control over the content generator must be balanced between the player and the game itself. When creating a PCG-based game, the role of the game designer shifts from being a creator of instances of content to an entire, parameterizable range of content. The challenge here comes in ensuring that, while the content the player experiences will be varied, the overall game experience is still appropriately controlled. For example, *Galactic Arms Race* retains control over when new weapons are evolved and seeds the pool with content that is known to be enjoyable.

Another method for balancing control over the content the player sees is to guide the player through the generative space with carefully placed goals. The goals in *Endless Web*—finding the dreamers and the powerups—are placed so that the player can experience the full range of content that Launchpad is capable of creating. Without these goals, the player's exploration of the generative space might never take the player to any interesting regions of the space, and the exploration goes unmotivated.

Keeping the PCG visible to the player. It is crucial for players to understand the effect they have on the world, and this can only be done by making the use and consequences of PCG as visible to the player as possible. Without knowing that the world is being generated around them, players lose agency as the choices they make feel meaningless. This is an issue that *Endless Web* still struggles with – creating a world that appears seamless can make it hard to understand that the world is being created around the player. While it is obvious on replay that the content is different, in *Endless Web* the changes from one layer of the generative space to another are intentionally subtle. The changes are obvious when the generator introduces a new type of challenge, but when the generator is configured at a layer that mixes all of the different challenge types, it can be hard to tell what has changed from one portal to the next. It is only when stepping back from the game and seeing different world configurations over time that the generator's role is obvious.

Building support for art. Building complete games requires effort from not only engineers and designers, but also artists and musicians. While procedurally generating art and music is possible and can lead to successful results (Ohkubo 2003; Persson 2011; Rohrer 2011), the resulting aesthetic is not always desirable. Thus, it is vital to build in support for art and

music to the PCG system, and design the system to be flexible enough to accommodate a changing art or music style over the course of the game's design. For example, Launchpad was designed to create levels for variably sized content, which was helpful when testing different size art assets for *Endless Web*. There were also rules in place for how content should be recombined at runtime, which helped both the artists and designers predict different potential configurations of individual scenes in the game. *Endless Web* also was designed to take advantage of the difficulties in art direction for procedurally generated content by having an intentionally jumbled world. There is a great deal of future work in "procedural art direction" for PCG-based games, including not only dynamic camera placement, dynamic lighting, and texture blending, but also aesthetic constraints over the structure of content.

8 DISCUSSION

There were two main goals when we began the design of *Endless Web*:

1. To evaluate Launchpad's usefulness as a controllable level generator for use in game design, and
2. To better understand the use of PCG in games and design a game that could not exist without PCG.

Overall, Launchpad's original design proved well-suited for PCG-based game design. That it could accept parameters regarding both pacing and geometry component composition was useful when balancing control over the generated content between the player and designer. These parameters and their direct impact on gameplay also led to a natural way to integrate

the generator into the game. However, creating *Endless Web* also uncovered a main weakness in the original Launchpad generator: a lack of perceivable variability in the difficulty of levels, and a tendency to create some common patterns for certain rhythm types. In early playtests of the game, players would remark that they felt like they were recovering old ground, when in fact they were in newly generated content. These two weaknesses led to the two major changes to Launchpad: introducing a tiered difficulty system, and relaxing some of the timing constraints to see more variety in generated content.

Endless Web is a game that is built entirely around the Launchpad generator; it could not exist without procedural content generation. There are many different configurations of parameters for the generator, and each configuration must have a large variety of content generated so that the player feels like they are exploring an endless world. This is far too much content for a team of human authors to create on their own without procedural support. The player can build strategies around the content generator, choosing to strengthen or weaken challenges along a particular axis in order to keep the content at a manageable difficulty level while exploring to find the dreamers and the powerups. *Endless Web* thus fits into a new genre of game—the PCG-based game. This genre is defined not by the setting or mechanics of the games it comprises, but the use of procedural content generation as a mechanic. In the future, we plan to release *Endless Web* to the general public so that we can learn further lessons from its reception and analyze common patterns in collected play trace data. It is my hope that the lessons learned in the creation of *Endless Web* can help drive the further creation of new PCG-based games.

CHAPTER 7

EVALUATING THE EXPRESSIVE RANGE OF PCG SYSTEMS*

In Chapter 2, I discussed the importance of controllability and variety in creating content generators for game designers. The ideal generator for design should be expressive enough to create a wide variety of styles of level, and have changes in those styles due to influence from the human designer be predictable. Evaluating a generator thus involves satisfactorily describing its ***expressive range***—the relationship between variety and controllability of the generator—according to design-relevant metrics. However, prior to the work described in this chapter, the majority of evaluation for content generators has involved either showing examples of generated content or providing player ratings of that content. There has been no effort to understand the expressive capabilities of a content generator or means for objectively describing the content that a generator can produce. This chapter describes two new methods for measuring and visualizing the expressive range of a procedural level generator, and applies them to the Launchpad and Tanagra generators.

1 EXPRESSIVE RANGE

A generator's expressive range describes the variety and style of levels that the system can generate and how sensitive that variety is to the input parameters for the generator. A

* The description of the expressive range techniques in this chapter, and their application to analyzing Launchpad, is modified from a Transactions on Computational Intelligence and AI in Games (TCIAIG) journal article (Smith et al. 2011b); an early expressive range analysis of Tanagra appears in another TCIAIG article, though that has been substantially modified and extended for this dissertation (Smith et al. 2011f).

future for PCG for design is one in which game designers, or even the players themselves, will want to choose different off-the-shelf level generators to include in their games. For example, one designer may wish to use a level generator that creates long, linear levels designed for speed runs, whereas another may want a generator that can create intricate, branching levels for the player to explore. These designers may be each able to use the same level generator that has a high expressive range that can be easily controlled, or they may each be better served by two different level generators that are specialized to each domain. For these hypothetical designers to make an informed decision about which generator they should use requires a clear way to communicate the expressive range and controllability of each generator they can choose from.

Understanding the expressive range of a level generator is also useful in driving future work for level generation. Expressive range analysis can uncover unexpected biases and dependencies in a particular generator, show weaknesses in the variety of levels that can be produced, and identify a need for more meaningful methods for classifying generated content.

The following questions are considered when judging the expressive range of these generators:

1. What are the appropriate ways to measure and compare produced levels?
2. How does the design of the generation algorithm itself affect the kinds of levels that can be produced?

3. What parameters should the designer expect to have control over, and how does altering these parameters impact the produced levels?

2 Measuring and Comparing Levels

In order to describe the expressive range of a level generator, it is first necessary to be able to judge the qualities of the levels that it produces. Some of these qualities can be determined through an examination of the implications from the generation algorithm itself, but others require a more scientific approach to measuring and understanding the generated content. This section describes the ways in which levels generated by Launchpad and Tanagra are analyzed for later visualization of their expressive range.

2.1 Algorithm Implications

Certain kinds of platformer levels are excluded from both Launchpad's and Tanagra's expressive range due to the design of the algorithms for level generation. Both systems create levels that are intended to produce a dexterity-based challenge rather than exploration. The challenge derived from these levels is more about perfectly timing movement through a series of obstacles than searching for hidden areas. Furthermore, neither system supports the player choosing a path through a generated level segment; any such exploration must be worked into the design of a game after generated sections have been made (as in *Endless Web*, Chapter 6). The level segments also do not provide any geometry for challenges that involve the player turning around. Launchpad and Tanagra both tend to favor creating levels more appropriate for a "speed run" play style, getting past challenging configurations of obstacles as quickly as possible while limiting damage to the player. This style influenced the choice of comparison metrics for levels, discussed below.

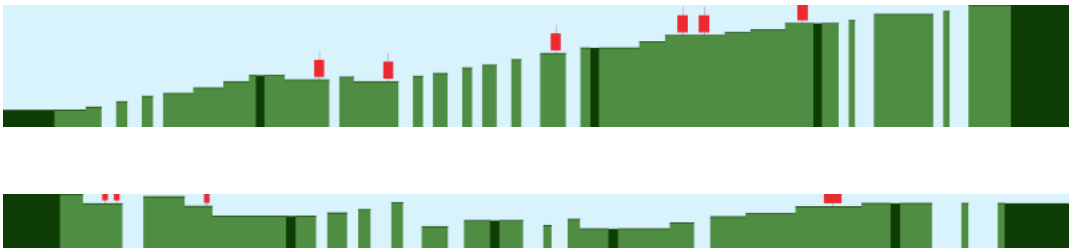

Figure 53. **Launchpad linearity example.** Examples of levels generated by Launchpad with the highest (top) and lowest (bottom) linearity scores.

2.2 Level Metrics

It is important that the metrics used for comparing levels measure emergent properties of the levels that are designed, rather than simply reusing the same input parameters used to guide the generator. By keeping the output measurements as separate as possible from the input parameters, it is possible to see how the input parameters impact the quality of the resulting levels. The two metrics used in the analysis provided in this chapter are based on the style of platformer that both Launchpad and Tanagra create levels for. Each of these metrics describes a global quality of levels, focusing on aesthetics (the linearity metric) and gameplay (the leniency metric).

2.2.1 Linearity

Linearity measures the "profile" of produced levels; this is a more aesthetic quality that the player will experience while rushing through the level. Linearity is measured 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 the generator's ability to produce levels that range between highly linear and highly non-linear. Examples of levels that fall at the extremes of this scale are shown in Figure 53. The linearity of a level is

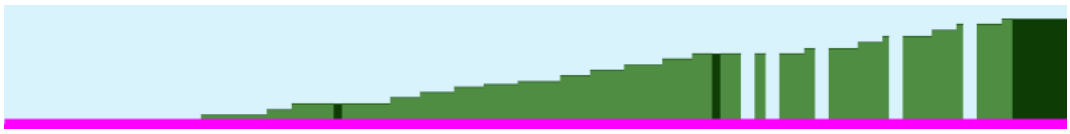

Figure 54. **Linearity vs. Line Critic.** An example of a level generated by Launchpad with a high *linearity* score but a low *line critic* score. The designer has requested that the level fit the line shown in pink (which remains flat across the whole level), which the level does not, hence the low line critic score. However, the level does fit well to a line that slants upwards, hence the high linearity score.

measured by performing linear regression, taking the center-points of each platform as a data point. Each level is then assigned a score equal to its R^2 correlation value, which will range on [0,1] where 1 is considered highly linear and 0 is highly non-linear (i.e., higher scores indicate a better fit to a line).

It is important to note that, in the case of Launchpad, linearity and the line distance critic are two different things playing different roles in the generation process. It is possible for a level to be judged highly linear but have a poor line distance critic score (see Figure 54). 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 a level that is output from the generator. Linearity is an aesthetic measure; line distance is a design control and heuristic.

2.2.2 Leniency

Leniency describes how forgiving the level is likely to be to a player. Leniency is a more objective measure than "difficulty"; difficulty is highly subjective, dependent on an individual player's prior experiences and on the specific ordering and combinations of components. However, it is reasonable to describe levels that provide fewer ways for the

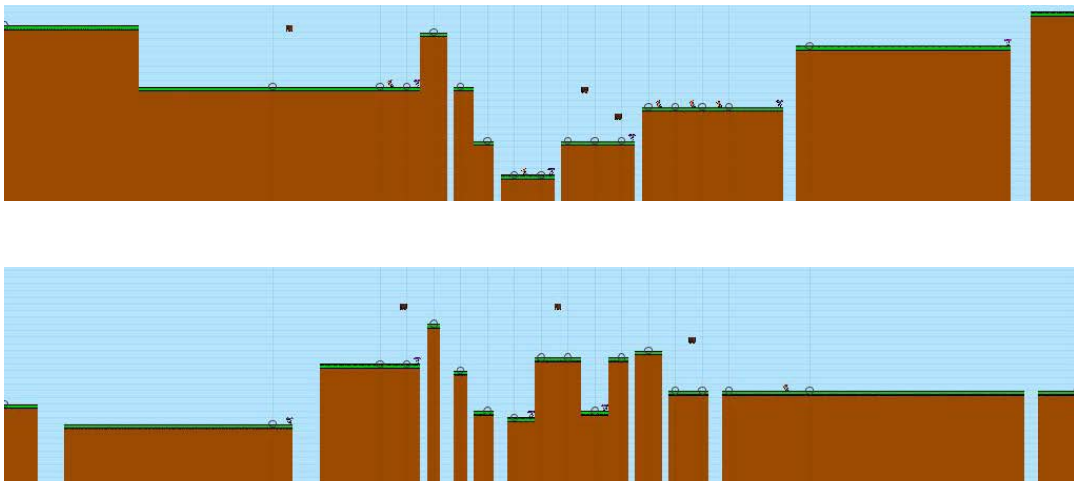

Figure 55. **Tanagra lenience example.** Two levels generated by Tanagra, with the same linearity score but different lenience scores. The top level has a lenience score of 0.61, the bottom level has a lenience score of 0.12.

player to come to harm as being more *lenient* than other levels. To measure this, scores are assigned to each type of geometry that can be associated with a beat:

0.0	gaps, falls
0.25	enemies
0.5	springs, stompers
0.75	moving platforms (where applicable)
1.0	jumps with no gap associated with them

These scores are based on an evaluation of how lenient the individual components are towards a player, with higher scores indicating greater lenience. The overall lenience for a level is calculated according to the following equation:

$$leniency = \frac{\sum_{b=0}^{n-1} time_b * geom\_score_b}{total\_time}$$

where *n* is the number of beats in the level, $geom.\_score_b$ is the weight associated with the geometry in that beat, *total_time* is the total length of the level, and $time_b$ is the length of time associated with the beat. In Tanagra, this is measured as the length of the beat that the component is in; in Launchpad, it is the amount of time between the last time the player completed an action and the time the player should take the new action. This means that non-lenient components are judged even more harshly if the player has little time to react to the challenge. Figure 55 shows two example levels generated by Tanagra with their associated lenience scores; lower scores indicate lower lenience.

2.3 Comparing Levels

Linearity and leniency both measure experiential qualities of entire levels. Another way for measuring expressive range is to judge the geometric similarity of content produced by the system. So, in addition to objectively ranking entire levels using these two metrics, we also define a method for directly comparing two rhythm groups according to geometric similarity. This distance metric is used for clustering rhythm groups to classify similarities and differences in the building blocks for 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 permits 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 (Levenshtein 1966) applied to vectors that code the type of geometry placed for each beat. This distance is normalized (Yujian & Bo 2007) to reduce the impact of the length of each

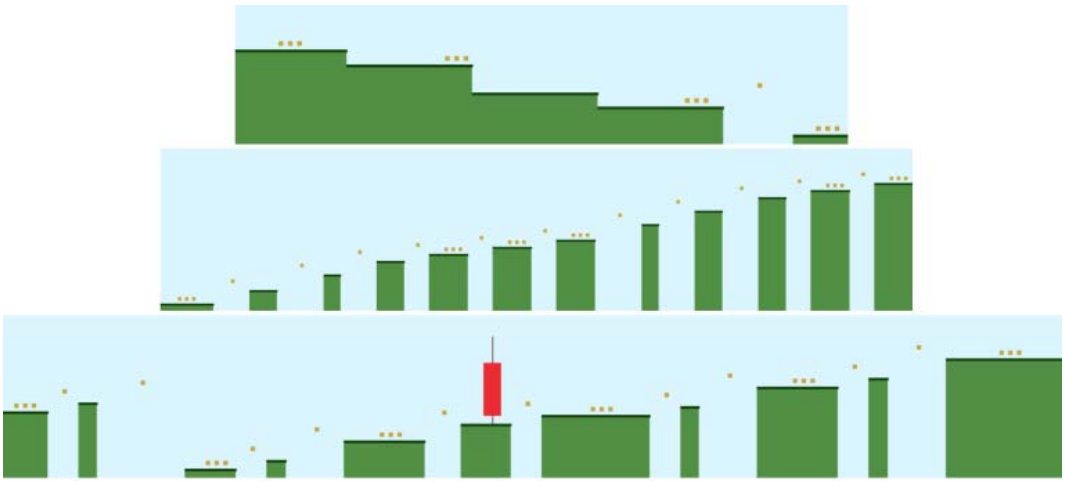

Figure 56. **Three rhythm groups created by Launchpad**, A, B, and C (top to bottom). The distances between these three rhythm groups are as follows: *d(A, B)* = 0.91. *d(B, C)* = 0.20. *d(A, C)* = 0.85.

rhythm on the distance between them. The distance metric defined here 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 of stomper

+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: 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, it is possible to extract emergent patterns (e.g. sequences of enemies) in the rhythm groups that are produced. Example rhythm groups and the distances

between them are shown in Figure 56. Notice that the distance between rhythm groups B and C is fairly low, even though they have different lengths and some different geometry. These two groups are more similar in terms of patterns than they are different.

3 Expressivity Analysis for Launchpad

The goal for understanding Launchpad's expressive range is to understand the relationship between input parameters and levels that are created, expose any biases or weaknesses of the generator, and provide a visual means for understanding the generator's expressive range that could then be compared to that of another generator. The methods for analyzing and comparing levels described in Section 2 permit a description of 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 visually portray the variety of levels that can be created with Launchpad.

3.1 Level Analysis

The first way for describing Launchpad's expressive range is by generating a large number of levels and ranking them by their linearity and leniency scores. Figure 57 shows the expressive range of the generator when all components are equally weighted and all rhythm types are being used. Each square in the plot is colored to indicate the number of generated levels that have the corresponding linearity and leniency scores. All graphs generated for Launchpad are based on 20,000 unique generated levels. A lighter colored square indicates there are more levels that fit in that linearity/leniency bucket.

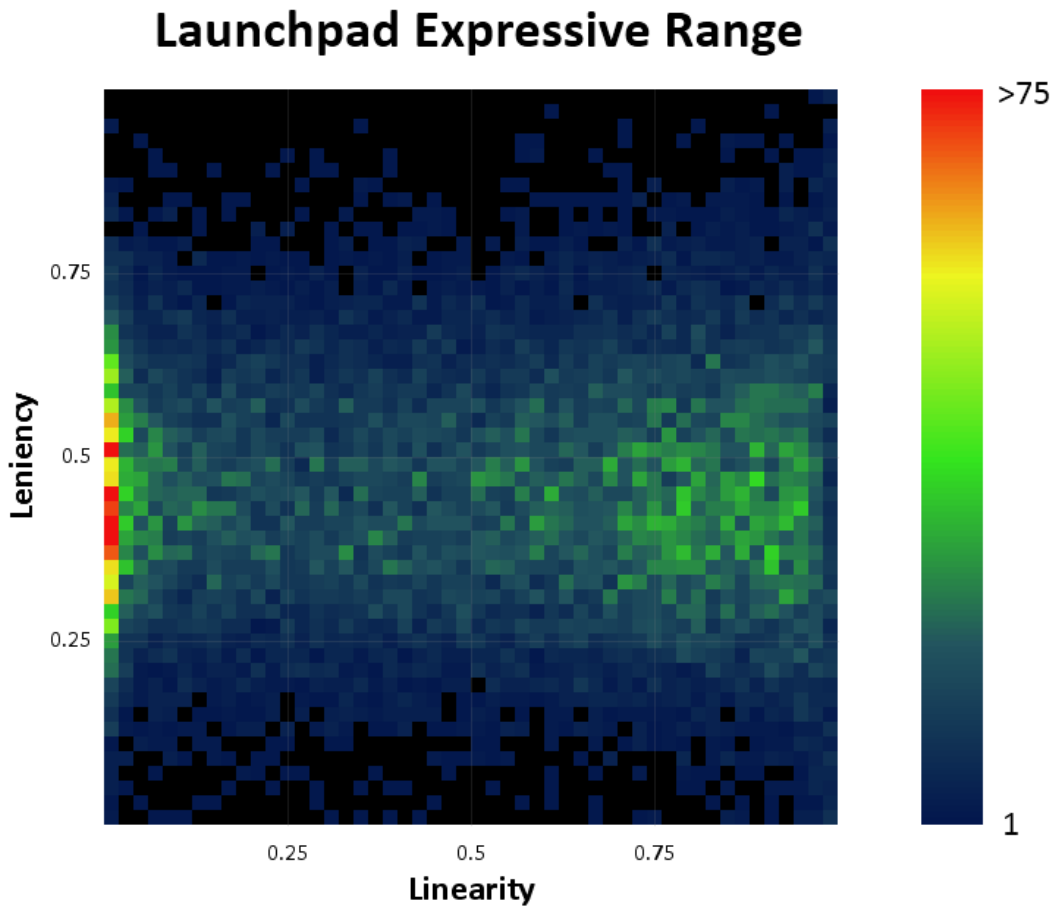

Figure 57. **Launchpad expressive range graph.** The expressive range for Launchpad when all input parameters are weighted equally. The gradient bar on the left shows the scale for all graphs in this dissertation: deep blue corresponds to a single level, bright red corresponds to over 75 levels.

There is good coverage of the generative space for both linearity and leniency when Launchpad is running with all its parameters weighted equally. The left side of the graph, where linearity is lowest, has a cluster of levels, which is to be expected given the variety of components that can lead to drastically changing y positions across the level. The right side of the graph, where linearity is the highest, shows a slightly wider spread of leniency; this is due to the influence of gaps on the leniency score. The good coverage of leniency 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

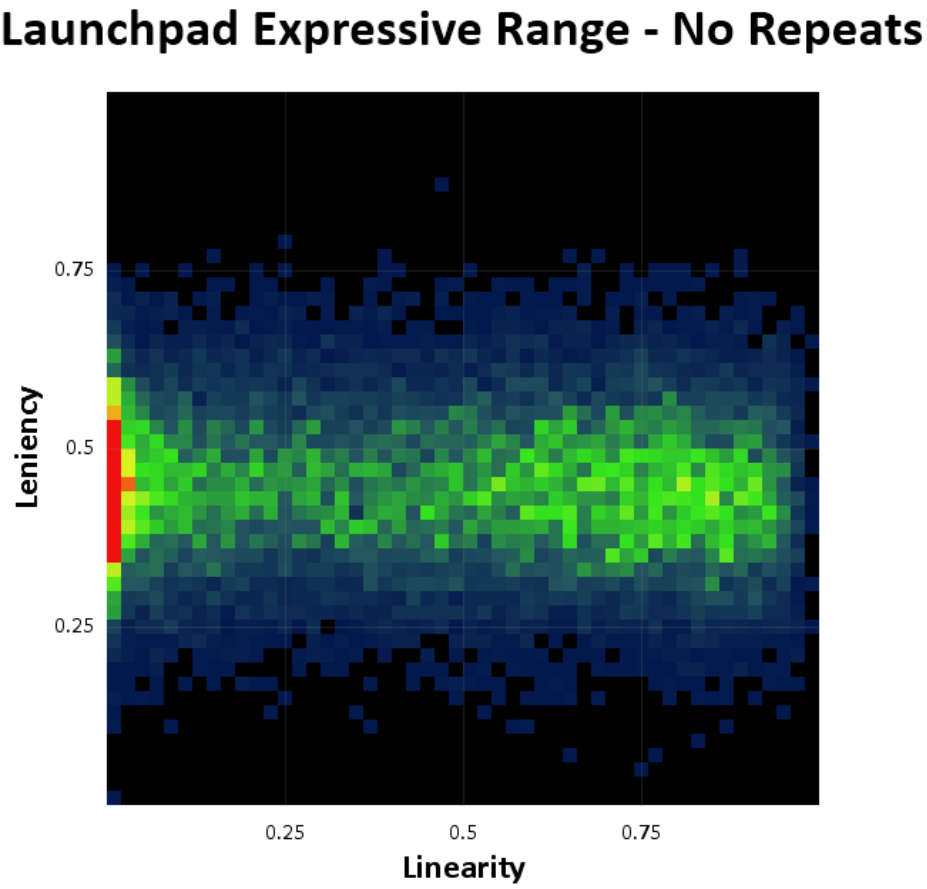

Figure 58. **Launchpad expressive range graph, results from disabling geometry repetition.** Launchpad's expressive range when the options for increasing previously chosen geometry and repeating rhythm groups are turned off.

component appearing again is very slightly increased. We also added a slight chance that previously chosen rhythm groups would be immediately repeated. These details were added late in the development of Launchpad to fix a problem perceived in the levels generated by earlier versions: they did not have any discernible patterns, as tend to appear in games like *Super Mario World* (Nintendo EAD 1990) where there tends to be locally repeated geometry.

The expressive range for Launchpad looks quite different when this detail is removed from the level generator (Figure 58). Note that the overall leniency range is greatly reduced. This

is because the repeating-geometry approach means that once a gap has been chosen, it is likely to be chosen again; conversely, once an element with no gap in it has been chosen, it is likely to be chosen again too. This means that greatly lenient and greatly non-lenient groups of geometry tend to stack up in rhythm groups, thus impacting the overall lenience of a level. Note also that there is a slightly higher density of non-linear levels when the repeat options are turned off; this is because there are far fewer stacked-up staircase patterns emerging from the generator. 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, it can be used to learn more about how rules interact inside the generator.

3.1.1 Expressive Range Biases

As described in Chapter 4, 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 impact the levels that are generated. For now, let us examine all levels that are created, rather than those that pass the critic tests. To uncover potential biases in the generator that emerge from parameter variations, we created expressive range graphs for each potential combination of rhythm parameters and examined the results for graphs that look substantially different from the expressive range when all parameters are weighted equally. Figure 59 shows each of these graphs side by side, in which it is possible to see high-level differences when varying parameters. This section describes some of extreme biases in the generator in more detail.

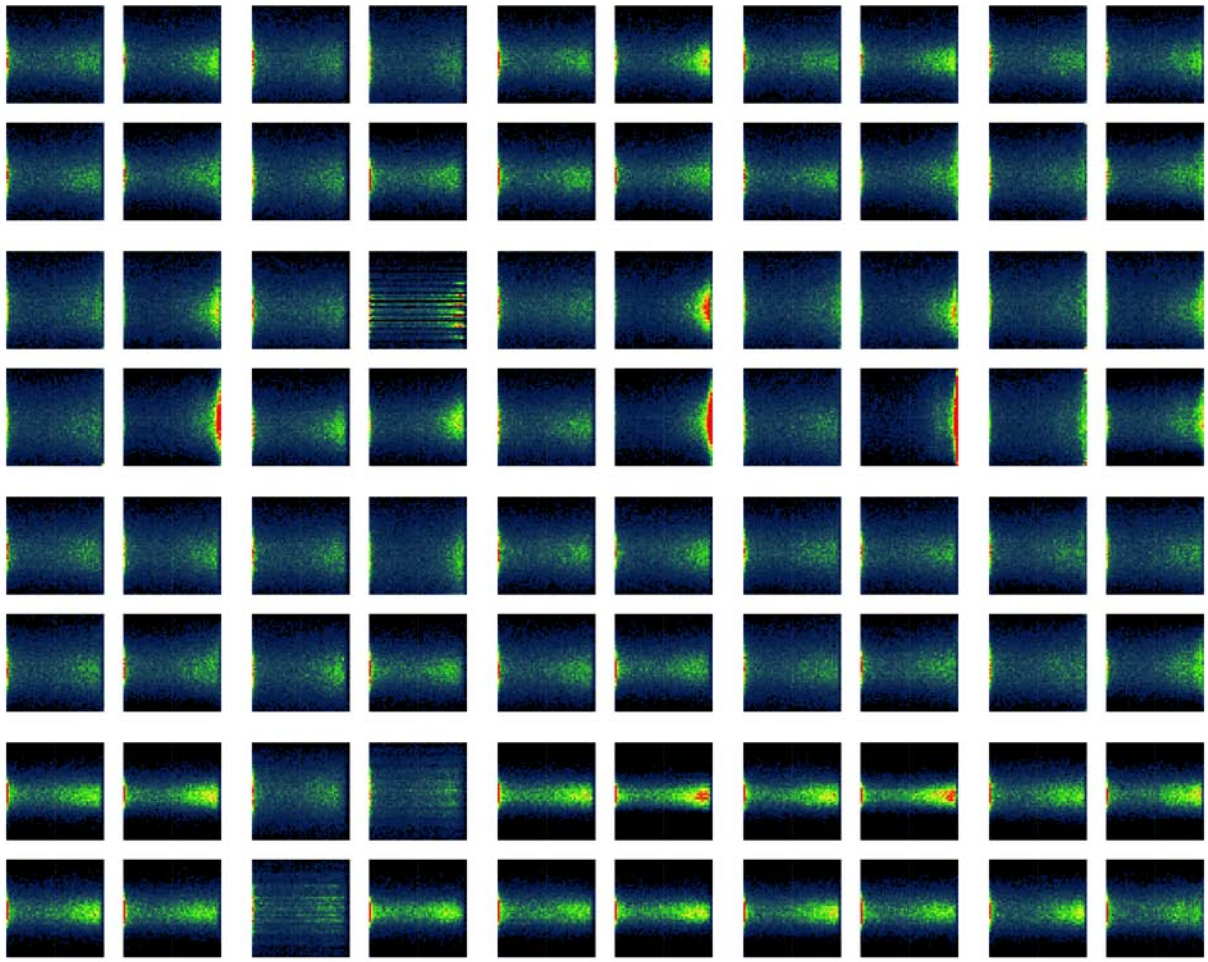

Figure 59. **Launchpad's expressive range graphs for each combination of rhythm parameters.** Graphs are arranged in clusters of four by row and column. Left to right: all rhythm lengths, length 5, length 10, length 15, length 20. Top to bottom: all rhythm types, regular type, random type, swing type. Within each cluster of four, clockwise from top left: all density, low density, medium density, high density.

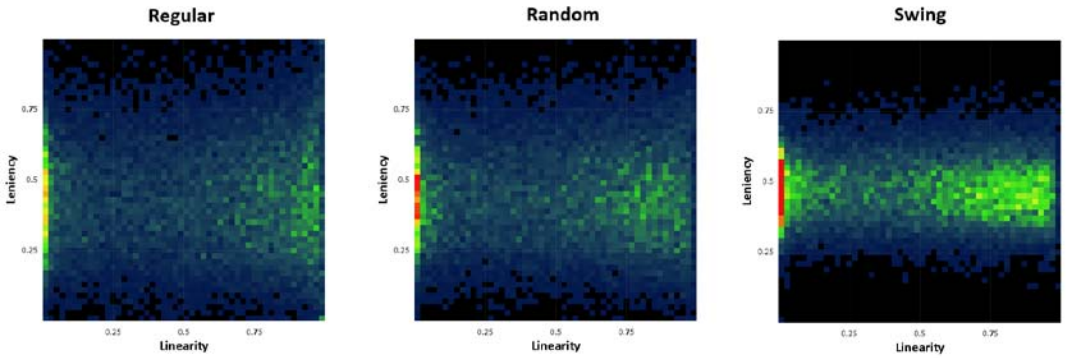

Figure 60. **Launchpad expressive range with varying rhythm types.** The expressive ranges resulting from varying only the **type** of rhythm used by Launchpad and leaving the rest of the parameters equally weighted.

Figure 60 shows the results of varying the rhythm type. The regular and random rhythm types offers the most variation of all the rhythm types, with no sharp peaks in the distribution. However, swing rhythms are more constraining, with far less of a range in leniency. One hypothesis is that these constraints are due to the potentially shorter amounts of time given to half of the jumps in swing 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 there will be fewer falls, meaning the overall leniency score will tend to be higher. The shorter jump times also means that many challenges will be judged more harshly, thus cutting out potentially highly lenient levels from the expressive range. Most combinations of parameters where a swing rhythm is required result in similarly narrow leniency ranges. The only exception to this is low and medium density short swing rhythms (Figure 61), since the number of beats in length 5 rhythms is so small that there is typically no impact on which jump types can be chosen (Figure 62).

[*] This aspect of the physics system was removed for the version of Launchpad used in *Endless Web* (Chapter 9) because the bias leads to a reduced variety in generated content when specifically requesting levels that contain a higher frequency of falls and springs.

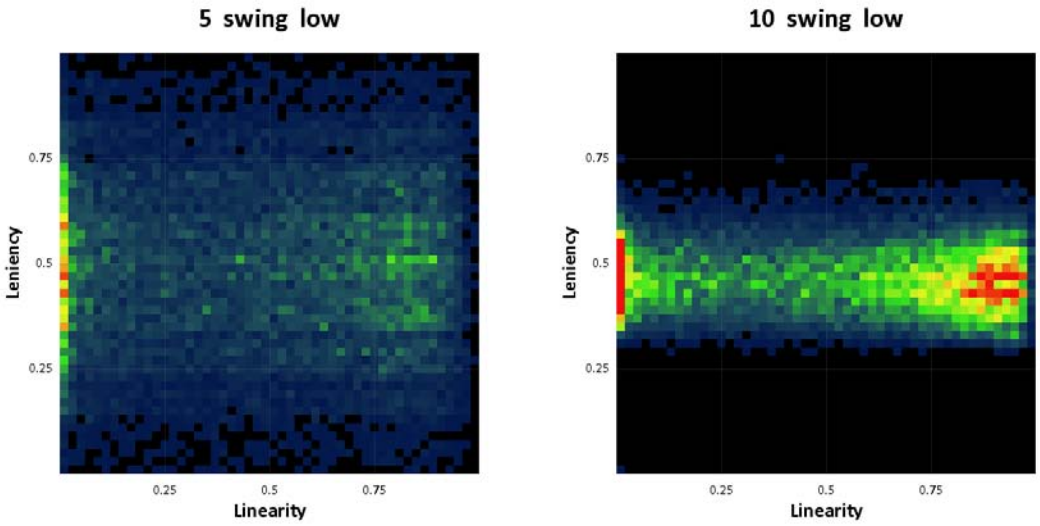

Figure 61. **Examples of Launchpad expressive range leniency biasing (graphs).** Expressive range graphs for Launchpad where the rhythm type is swing and the rhythm density is low. The left shows rhythm lengths of 5 seconds long, the right shows rhythm lengths of 10 seconds long. The graph for length 5 medium density swing rhythms looks similar to the graph on the left.

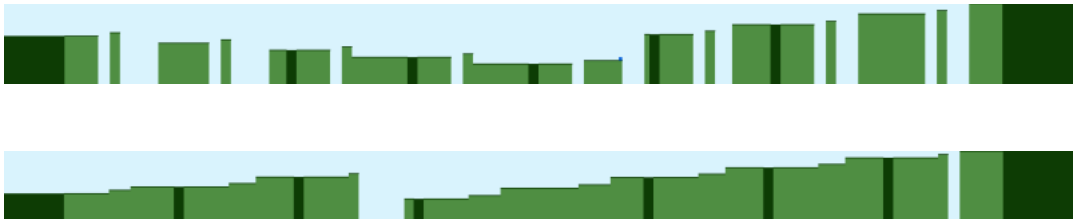

Figure 62. **Examples of Launchpad expressive range leniency biasing (levels).** Two Launchpad levels generated with parameters length=5, type=swing, density=low. The top level has low leniency due to the large number of gaps spread throughout the level, the bottom level has high leniency, as there are fewer ways for the player to come to harm.

There is also some biasing in the generator for low and high density, mid-length regular rhythms. In these cases, the generator is extremely biased towards creating highly linear levels; and levels that are not linear have a narrower leniency range. This is accounted for by the other reasons for biasing seen so far: the probabilities for repeated geometry and the need to filter out certain geometry components due to physics constraints. Removing the probabilities for repeated geometry shows us the impact that the physics constraints alone have on the generator's expressive range for this combination of parameters (Figure 64).

While there is better coverage for linearity of levels, the leniency range drops substantially (as expected), and there is still some biasing towards more linear levels.

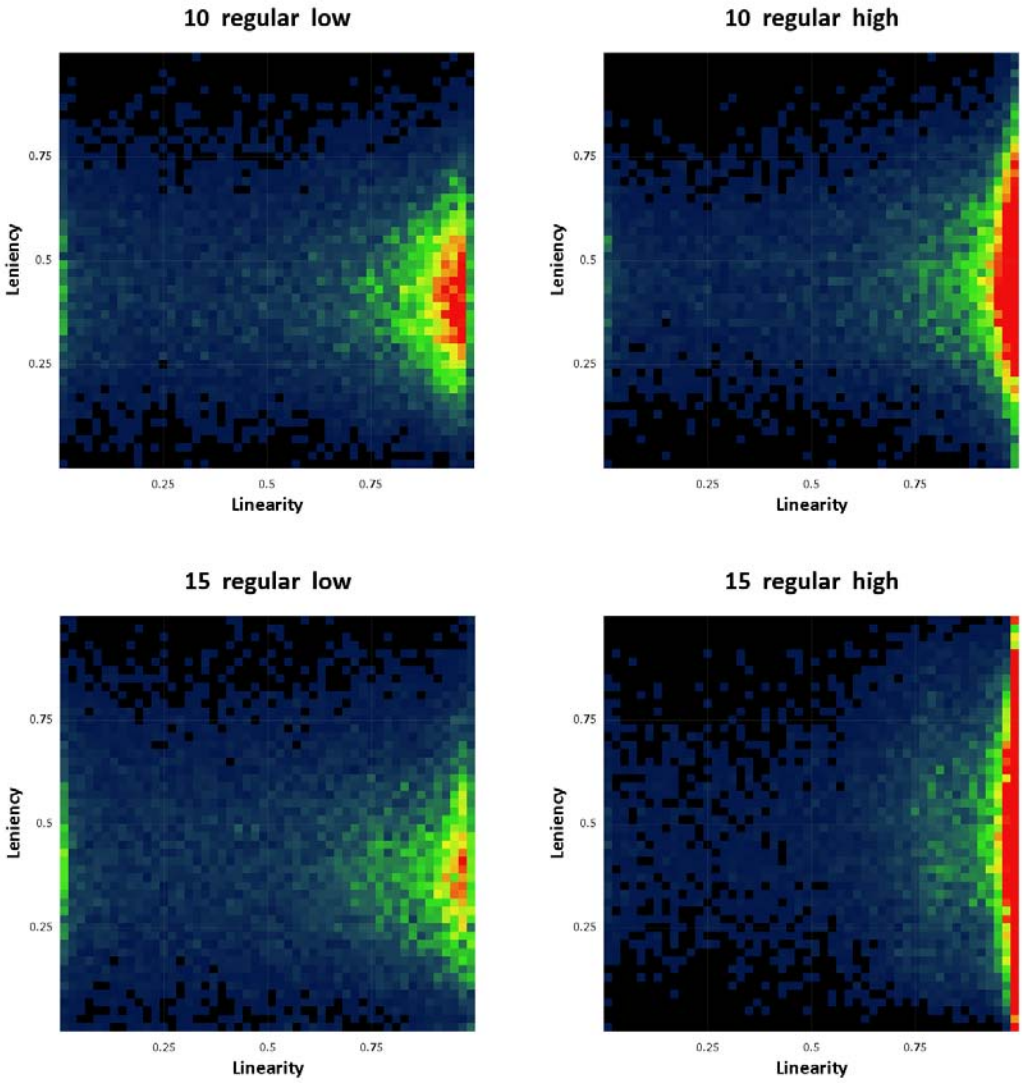

Figure 63. **Examples of Launchpad expressive range linearity biasing (graphs).** Expressive range graphs for length 10 and 15, low and high density, regular rhythms. The generator is biased towards creating more linear levels, and the non-linear levels it creates have a narrower leniency range.

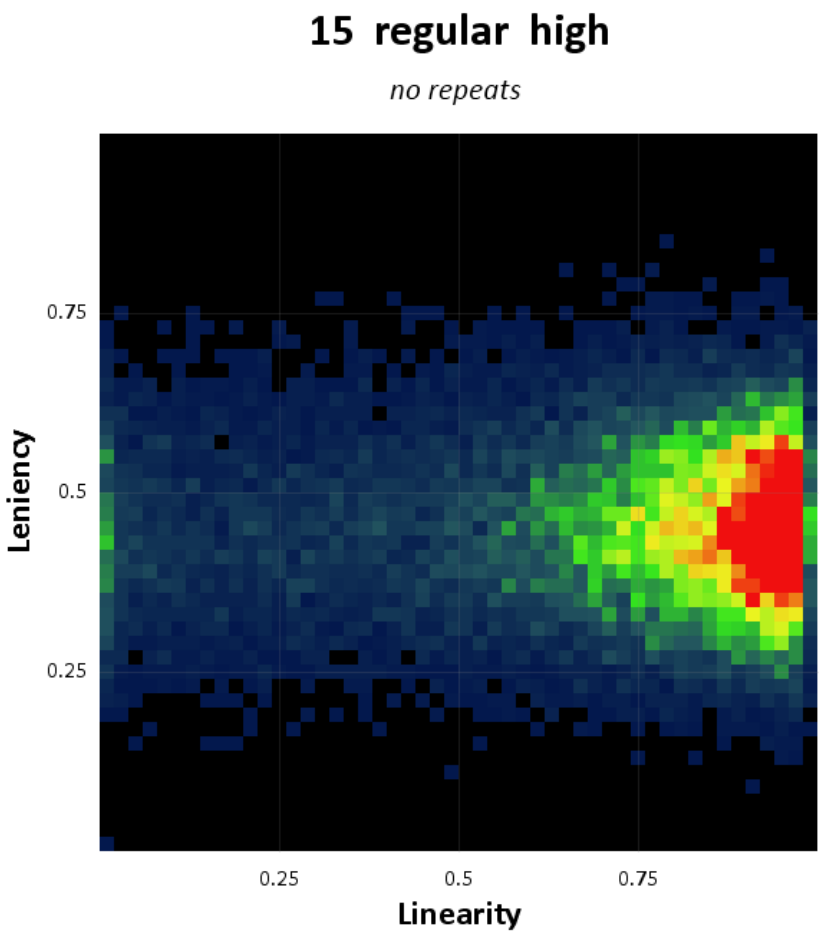

Figure 64. **Impact of removing geometry repetition on Launchpad's linearity-based expressive range.** The expressive range of Launchpad with parameters length=15, type=regular, and density=high, but without any probability for repeating geometry.

This is accomplished through a clustering of rhythm groups based on their similarity to each other, then visualizing these clusters in an interactive application to allow potential users of the generator to explore the kind of content they can expect to see.

Rhythm groups are clustered using an agglomerative hierarchical clustering method (Manning et al. 2008). 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

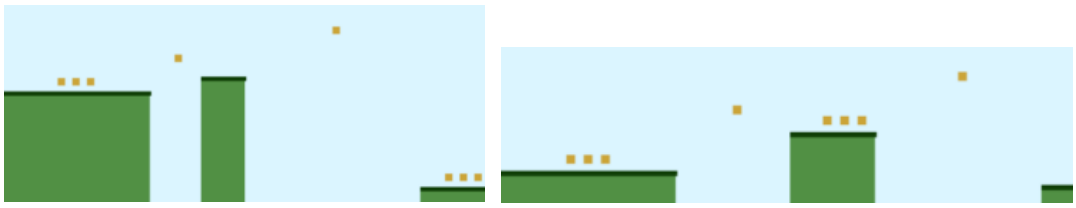

Figure 65. **Two Launchpad rhythm groups whose edit distance is zero.** Note that these groups are not identical, but do share the same pattern of jumping up over a gap, and then jumping down over a gap.

example of two rhythm groups that have zero distance between them is shown in Figure 65. These bins of similar rhythm groups form the leaf nodes of the hierarchy, and are the initial clusters provided to 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 dendogram; 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 the level designer navigate this tree using a treemap representation (Fry 2008), where each box has a size corresponding to the number of rhythm groups contained in the cluster. Each cluster also 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 was performed on 4000 rhythm groups: 1000 for each length of rhythm. 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 in the cluster, allowing easy visualization of variety within a single cluster. Figure 66 shows screenshots from 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 large number of stompers. There are also a large number of rhythm groups that contain geometry that forms an upward staircase pattern[†]. The discovery of this 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 from Chapter 4 that the wait-move-wait pattern also contains a wait-move, meaning the stomper can be chosen in both situations. This biases the generator towards creating a larger number of stompers than expected.

[*] The cluster visualization tool is included in the supplemental files accompanying this dissertation, and also available online at: http://sokath.com/dissertation_supplement/launchpad/viztool/

[†] Note that this expressive range analysis on Launchpad was performed prior to the development of Tanagra's multi-beat patterns (Section 2.2), which include a staircase pattern. The inclusion of multi-beat patterns was partially inspired by the discovery of common patterns in Launchpad.

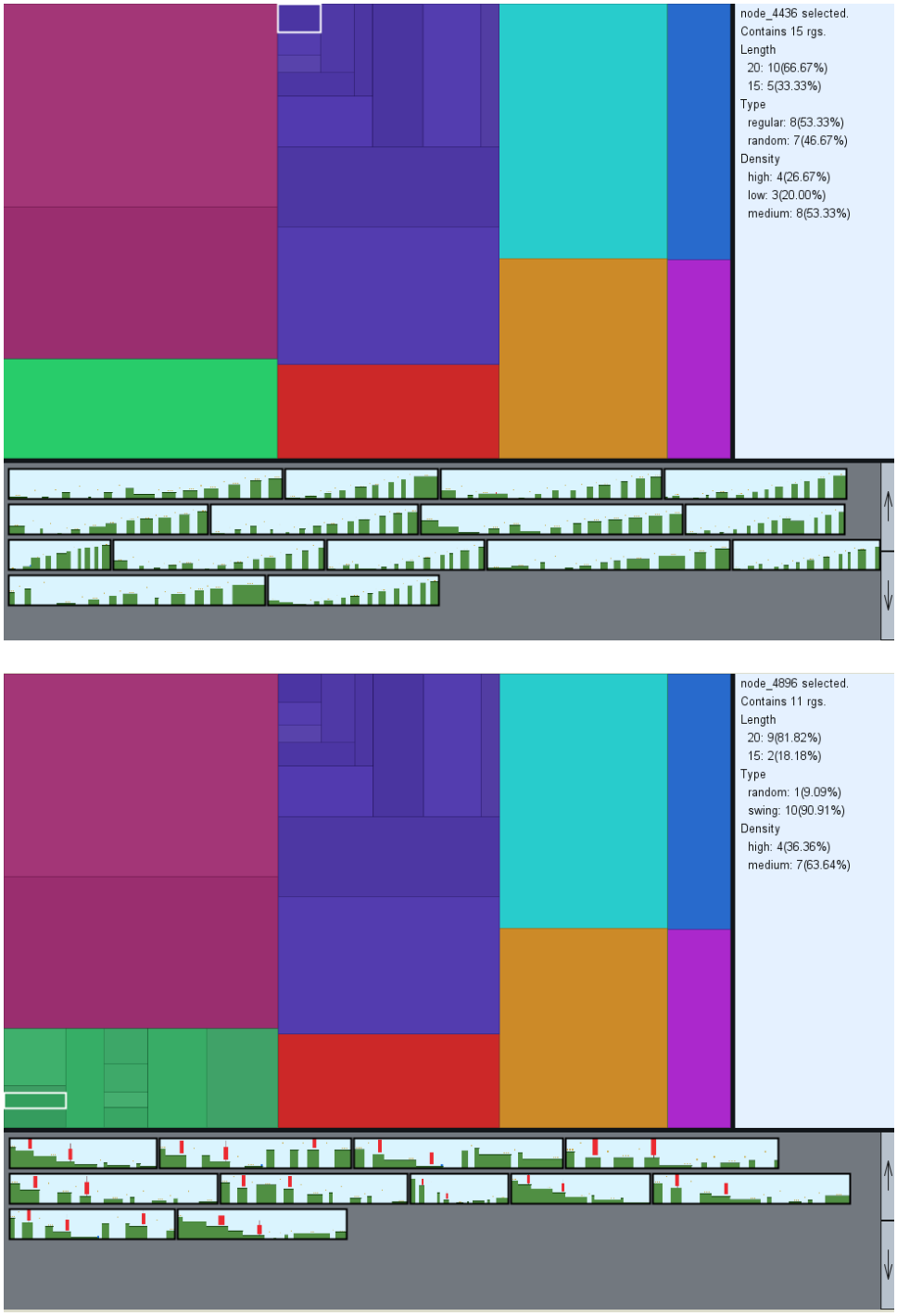

Figure 66. **Two screenshots of the rhythm group cluster visualization tool, showing two different highlight clusters.** The cluster in the upper screenshot shows a cluster of rhythm groups consisting of platforms with gaps in between them that tend to dip downwards in the middle. The cluster in the lower screenshot is of rhythm groups that form a descending staircase punctuated by stompers.

4 Expressivity Analysis for Tanagra

As a level generator that must support designers, it is especially important that Tanagra can create a wide range of levels. The human designer is entirely limited in what they can create by the capabilities and biases of the generator underlying Tanagra; it is impossible to create a level in the tool that the generator would not be able to make itself. Even a level that has been completely authored by the human must still be able to be supported by the underlying generator. The expressivity evaluation of Tanagra* uses the same technique as used in measuring the linearity and leniency of levels for Launchpad described in Section 2.1.

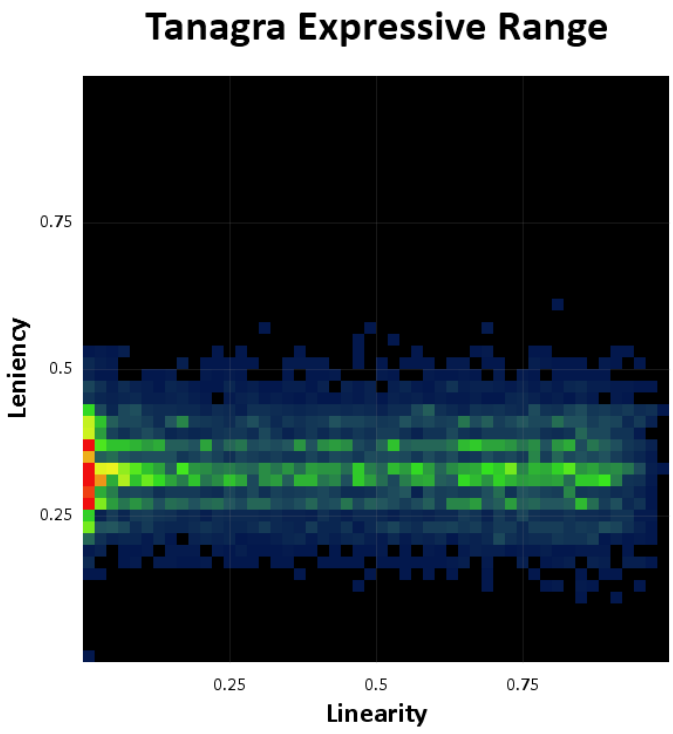

Figure 67. **Tanagra expressive range graph.** The expressive range graph for Tanagra with its default beat timeline and no user constraints.

* Note that this evaluation is performed using the newer prototype of Tanagra, as it provides more control options to the designer: placement of platforms, manipulation of the beat timeline, and specification of preferences for particular geometry patterns in beats.

Figure 67 shows the expressive range of the generator when it runs without any user constraints and with the default beat timeline. This data is collected by generating 10,000 levels without any user-placed geometry or modifications to the default beat timeline. Tanagra's expressive range is skewed slightly towards creating non-linear levels, and by default it only creates levels in a very narrow range of leniency. Tanagra's expressive range is quite different from Launchpad's, especially with regard to leniency. The reason for this is that Launchpad explicitly models jumps with no gap associated with them, whereas Tanagra groups jumps up onto new platforms with gaps. Therefore, there are far fewer incidents of a jump with no gap associated with them, which is weighted as highly lenient. Figure 68 shows a level that fits into the largest bin in Tanagra's expressive range, while Figure 69 and Figure 70 show examples of types of levels that occur less frequently.

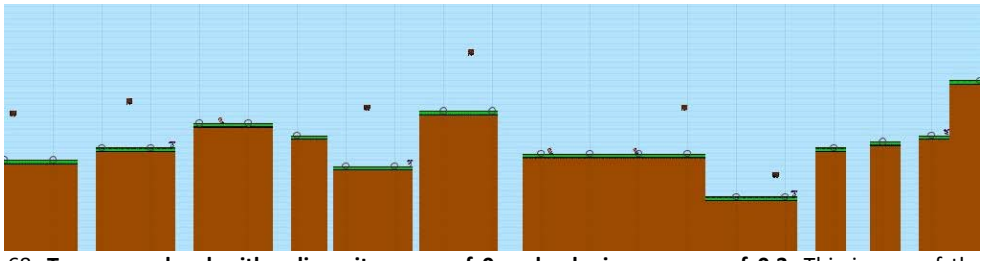

Figure 68. **Tanagra: a level with a linearity score of 0 and a leniency score of 0.3.** This is one of the most common kinds of levels that Tanagra creates.

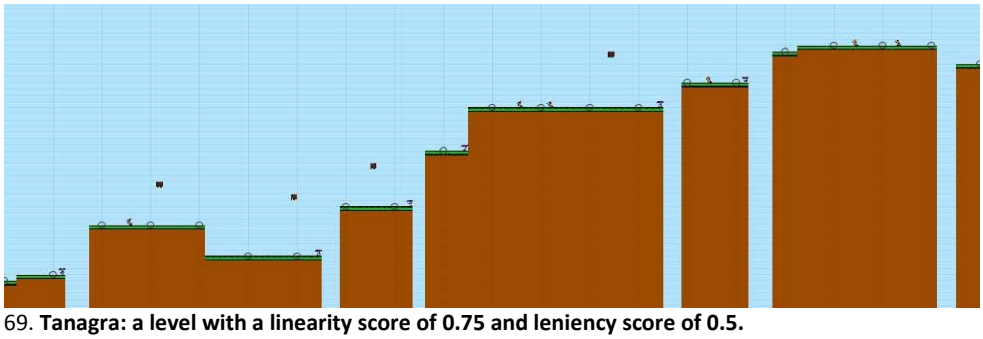

Figure 69. **Tanagra: a level with a linearity score of 0.75 and leniency score of 0.5.**

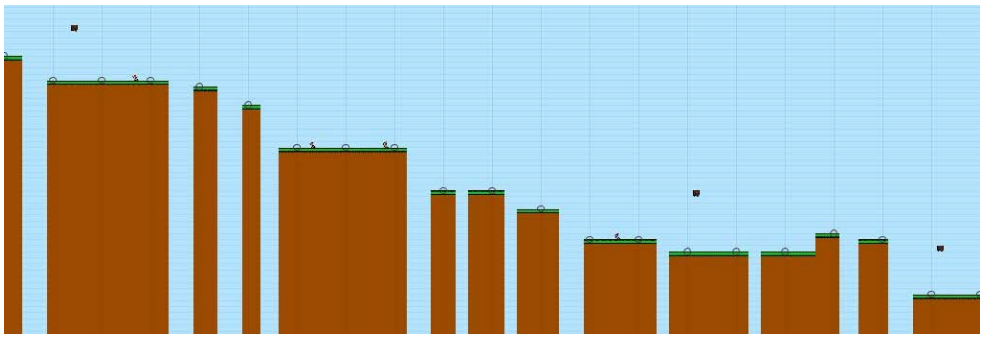

Figure 70. **Tanagra: a level with a linearity score of 0.9 and leniency score of 0.1.**

4.1 Shaping the Generative Space

However, it is not enough to judge Tanagra's expressivity solely on the qualities of levels it can produce in its default mode. It is important for a mixed-initiative tool to provide a controllable variety of levels throughout the editing process. This section investigates Tanagra's expressivity when being used as a tool, by showing how the shape of its generative space shifts while performing a series of level editing operations. This analysis uses the newer prototype version of Tanagra, as it provides the designer with more level editing operations than the prior version. The three main editing operations are editing the pacing of the level, moving geometry around the level, and declaring preferences for geometry patterns on a per-beat basis. In this evaluation, our hypothetical designer makes a number of changes to the level:

1. After noticing that the level is more slowly paced than desired, she alters the pacing to be faster in the middle of the level by splitting beats 8 – 14 in half.
2. With the pacing curve still not quite right, the designer decides to alter the beginning of the level to provide a more gentle introduction and conclusion to the level, with a relatively faster paced middle. She accomplishes this by deleting the second beat and the second-to-last beat five times each.

3. Now the designer chooses to experiment with different geometry options. She changes the geometry preferences to prefer springs in beat 2, enemies in the first three short beats, a sequence of stomper – fall – stomper in the last three short beats, and gaps in the second to last beat. She chooses to disallow gaps in all other short beats.
4. Unsatisfied with these results, she decides to again alter the pacing of the level to have two clear sections with rapid and more difficult challenges, and make the rest of the level gentler for the player. She deletes the middle five short beats, sets a preference for gaps and falls in the short beats, and disallows gaps and falls in the non-short beats.
5. Finally, the designer decides to alter geometry positioning to make the player feel like they're falling down into a pit and them climbing out again. She drags geometry so that the first third of the level is in the middle of the canvas, the middle third is towards the bottom of the canvas, and the right third is towards the top of the canvas.

During the course of editing this level, Tanagra's expressive range changes quite a bit – opening up new regions of the space for editing and confining others. Importantly, even with the most restrictive changes made by the user (dragging geometry and setting geometry pattern preferences), Tanagra still displays a wide range of content that can be created to support the designer's choices at each step along the way, but it is slowly being guided towards a designer-targeted space. Figure 71 shows the six different expressive range graphs that correlate to the initiate state of Tanagra and the stages of editing described above, alongside representative levels associated with the editing process. Here we describe the changing expressive range and show an example level produced at each stage of editing.

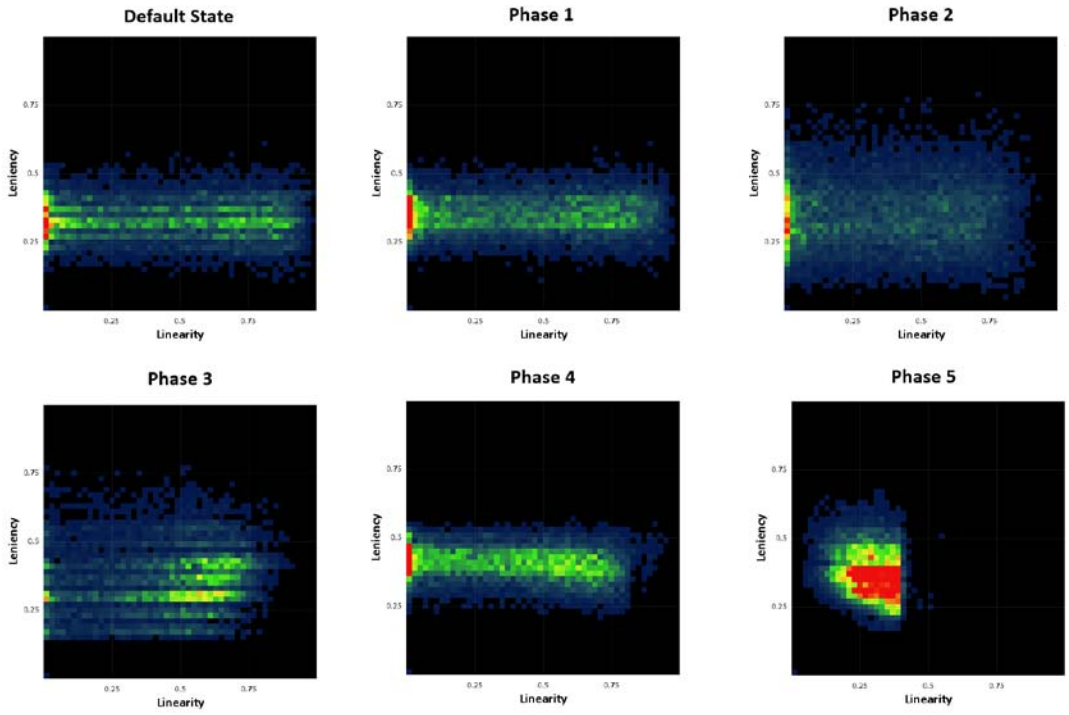

Figure 71. **Tanagra's expressive range, changes due to a level editing scenario.** Tanagra's expressive range in its initial state, then after each of the 5 phases of editing described in this use scenario. At each stage of editing, the designer is shaping Tanagra's expressive range in a predictable way, while Tanagra is still showing that it is capable of producing a variety of levels within the parameters associated with the designer's edits.

In the first stage of editing (Figure 72), Tanagra's expressive range does not change a great deal. The leniency shifts upwards very slightly, likely due to the increased number of opportunities for lenient patterns to be included in the level.

The second stage of editing (Figure 73) shows a larger increase in Tanagra's expressivity in terms of the leniency of generated levels—it is now capable of making both more and less lenient levels than before. While there is still a cluster of levels that are more likely to appear, that cluster is larger and less concentrated. There is also a shift away from the most linear levels; this is due to the long beats at the beginning and end of the level, which encourage long flat areas in the level.

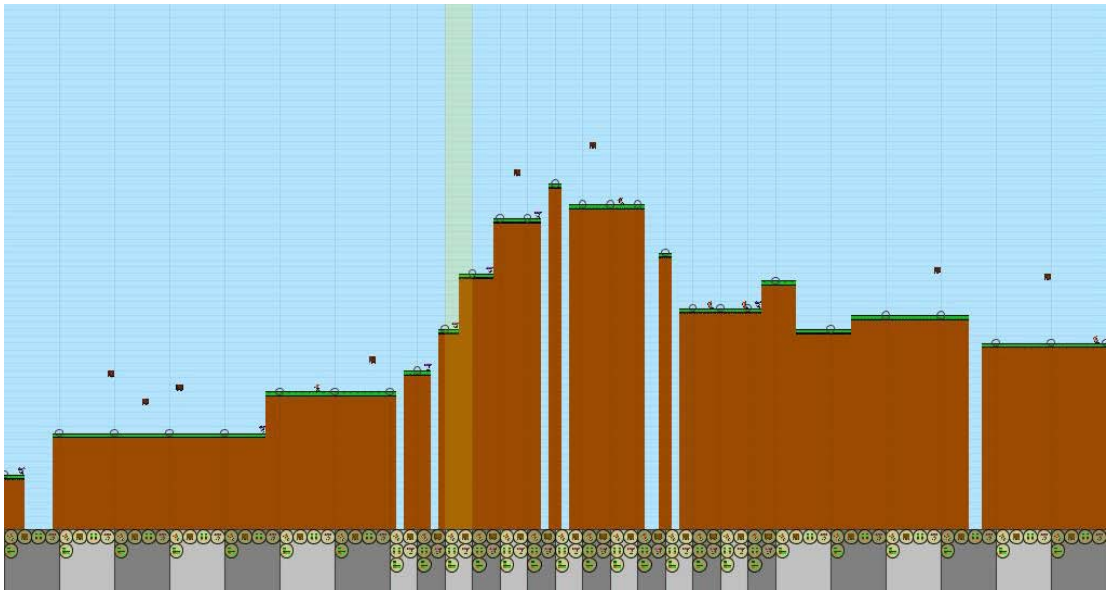

Figure 72. **An example of a level produced by Tanagra after the first phase of editing.** There are more, and shorter, beats in the middle of the level.

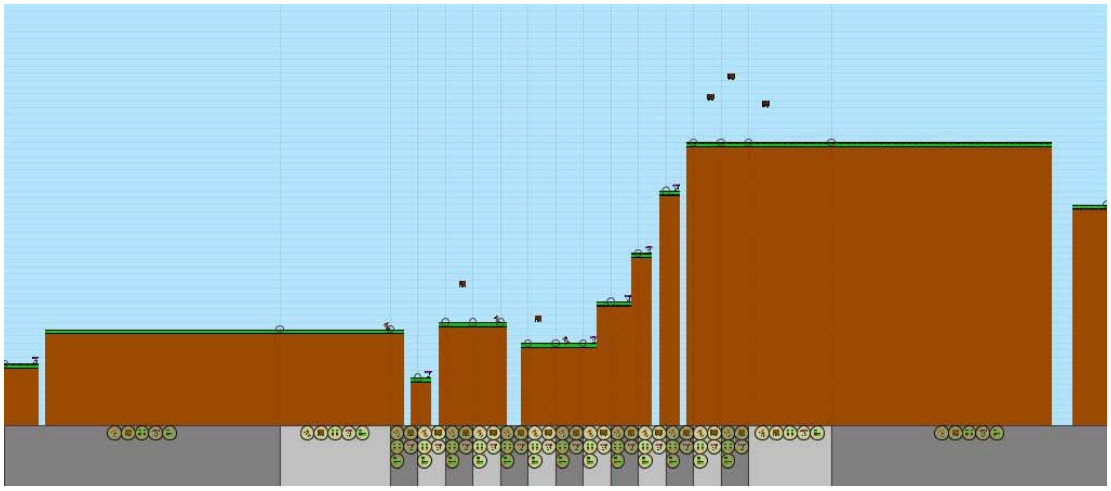

Figure 73. **An example of a level produced by Tanagra after the second phase of editing.** There are now long beats at the beginning and end of the level, forcing flatter geometry there than elsewhere, thus removing the likelihood of making the most linear levels.

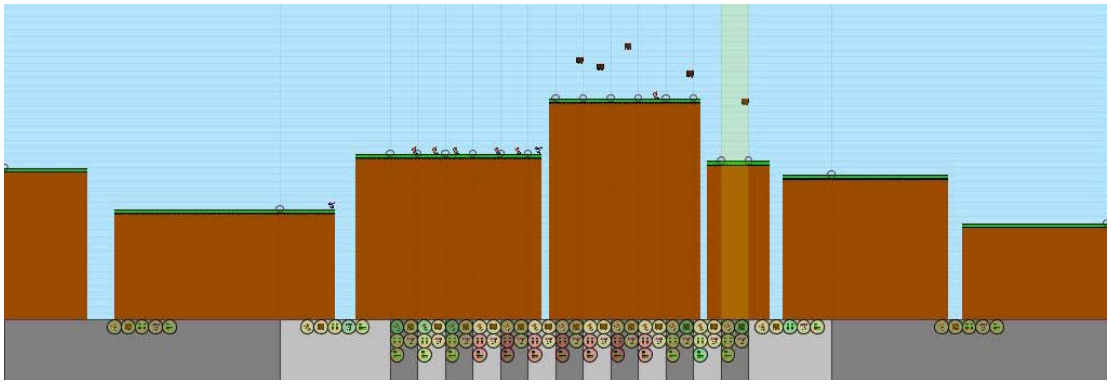

Figure 74. **An example of a level produced by Tanagra after the third phase of editing.** The designer sets preferences for geometry in phase 3 of editing that tend to increase the appearance of stompers and enemies in the level.

Figure 75. **An example of a level produced by Tanagra after the fourth phase of editing.** The designer changes the pacing model in the level and the geometry preferences so that short beats should contain gaps and long beats should not.

Disallowing gaps in many of the short beats in the third stage of level editing adjusts the generator's expressive range to no longer include the least lenient levels it could create in stage 2 (Figure 74). Preferring gaps in short beats in the fourth stage of editing has the opposite effect, removing a large chunk of lenient levels from the previous stage's generative space (Figure 75).

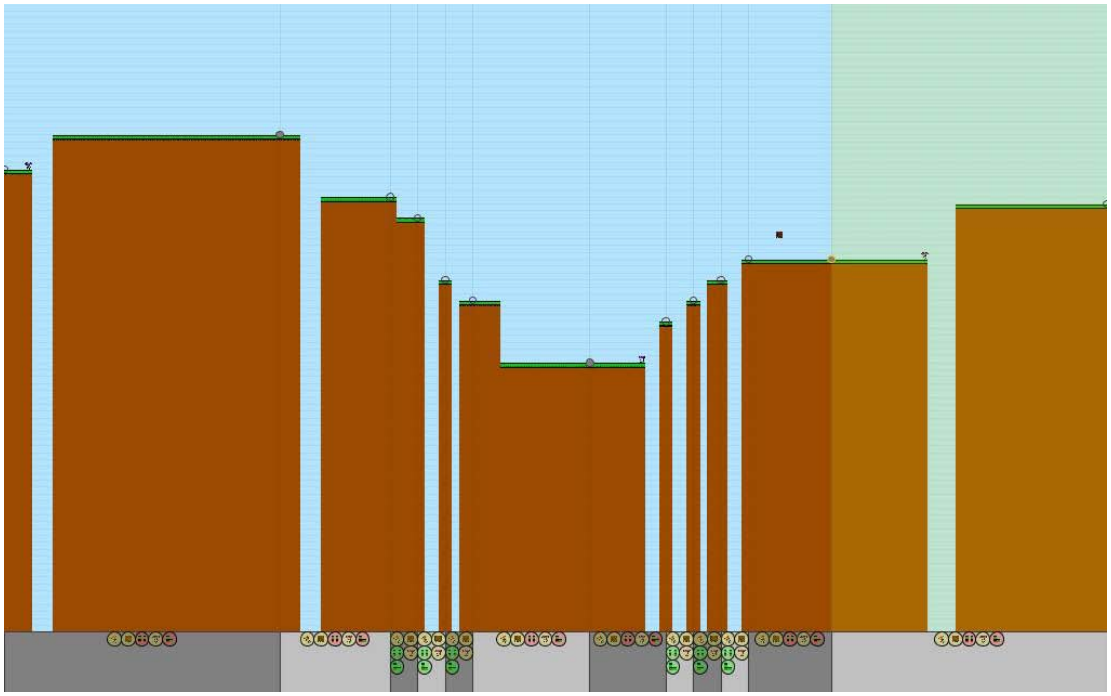

Figure 76. **An example of a level produced by Tanagra after the final phase of editing.** In the final phase of level editing, the designer pins the end of the first beat to be higher up than the middle of the level, and the beginning of the final beat to be the middle y position between the start and middle of the level.

Finally, the way in which the designer moves platforms in the last stage of editing forces Tanagra into creating only non-linear levels (Figure 76).

Overall, this analysis of Tanagra's expressive range throughout the editing process shows that the designer's actions continually shape the generative space of the system in a reasonable and predictable way. Tanagra is capable of supporting a designer exploring different gameplay options for a level design (through pacing and geometry pattern changes) by providing an environment in which it is guaranteed that levels will be playable, yet still offering sufficient variety in the kinds of levels that can be created in each phase.

5 DISCUSSION

Understanding a generator's expressive range is crucial for the ability to create generators for use in game design. Generators must be sufficiently controllable that the results of a designer's actions are reasonable and predictable, while still offering enough variety that the designer can easily play with different ideas for a level, or the player does not see patterns of repeated geometry configurations. The work presented in this chapter offered two new methods for understanding and visualizing these capabilities of a generator, providing the ability to characterize the expressive range of generators and the impact of design choices on their output. This allows users to understand the abilities and limitations of a particular generator, and the designer of the PCG system itself to better understand any biases that arise from the system's design. It is my hope that these methods, and others like them, become a primary method for evaluating PCG systems in general.

When using PCG in the game design process, the designer will always be limited by the expressivity of the content generator. In Tanagra, a high expressive range is important because the best level a designer can create is still one that the system must be able to generate itself. *Endless Web*'s entire game design relies on the player being able to control the generator but still see a wide variety of content for each set of potential input parameters.

There is a great deal of future work in the area of analyzing the expressivity of content generators. A primary area for improvement is in determining better metrics for comparing levels. This is especially important when considering that a motivating factor for this work is being able to quantitatively compare two different level generators. While we feel that

linearity and leniency are important metrics for comparing platformer levels, they are not sufficient on their own and there is plenty of room for new and improved metrics. Solely aesthetic measures do not seem sufficient; a model of player behavior, perhaps encompassing different play styles, would be an interesting way of attacking this problem. Difficulty is an obvious measure, but other measures may include the perceived challenge due to pacing variation across the level, or different camera positions.

There is also room for improvement in being able to use expressive range as feedback to a designer in a tool such as Tanagra. Understanding how incremental changes to a level are impacting the overall player experience is crucial feedback for a designer. Being able to see where in the generative space a level currently is, and perhaps more importantly what areas of the generative space have been cut off due to a recent change, is an important area of future work in creating more intelligent design tools. The methods for providing this feedback must be substantially different from those presented in this chapter, however, as generate-and-test is a time-consuming process and the feedback to the designer should be provided in realtime.

Finally, this work has only scratched the surface of the application of visualization techniques to expressive range analysis. More sophisticated metrics for rating generated content and a desire to provide realtime feedback to a designer will require more innovative ways to present salient information about the generator's expressivity to both its creator and its users.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK*

1 SUMMARY OF CONTRIBUTIONS

This dissertation has focused on the use of procedural content generation to create **expressive design tools**, which I introduced in Chapter 1 as:

> **Generators imbued with an understanding of a game's design that are sufficiently controllable and expressive for use in the design process.**

The creation of these tools requires a new approach to procedural content generation, one in which the PCG system is imbued with an understanding of a game's design and intended player behavior, and that can maintain an appropriate level of both controllability and variability. In the pursuit of expressive design tools, I have made six major contributions:

1. A taxonomy of current PCG systems in terms of their suitability for use in the design process, covering the ways in which generators can be controlled and the extent to which that control plays a role in the final produced content.

2. A rhythm-based representation for 2D platformer levels that makes it possible for a content generator to reason about level pacing.

* Section 2.1 of this chapter has previously appeared in the 2011 Workshop on Semi-Automated Creativity (Smith et al. 2011e).

3. The first autonomous, parameterized level generator, named Launchpad, that operates in the domain of 2D platformer levels and can accept design parameters related to both player experience (in the form of desired level pacing) and the composition of geometry components.

4. Tanagra, the first ever mixed-initiative level design tool that allows a designer to exert direct control over gameplay-relevant aspects of a level's design. Tanagra is also one of a handful of mixed-initiative design tools in existence, and the first to allow a designer a means for interacting with a procedural content generator.

5. *Endless Web*, a game that inhabits a new genre of games called PCG-based games. It is the first game to give a player explicit control over an underlying PCG system, and it is designed to have the generator heavily tied to both its mechanics and aesthetics, such that the player forms strategies around the generator. *Endless Web* uses the Launchpad level generator to create an infinite world for players to explore and manipulate.

6. A new approach to evaluating procedural content generators, called expressive range evaluation. This evaluation involves measuring and visualizing the full range of content that a generator can create, including how the range changes based on different design inputs, permitting an examination of the tension between controllability and variability in content generators.

The remainder of this chapter outlines my thoughts on future research in PCG for game design. It is my hope that the work presented in this dissertation will guide new research in procedural content generation as it applies to Juul's "rules" half of game design (Juul

2005)—building content generators that can reason about the underlying systems in games and how those systems influence player behavior. However, I do not wish to downplay the importance of game's fictional worlds; the future of procedural content generation for game design also lies in creating systems that can bridge the two halves—rules and fiction—of game design.

2 Future Work

There are many potential future paths that the work described in this dissertation could follow. Some of these continue with the primary motivation for this work: developing technologies that enable new kinds of games and design experiences. Others delve more deeply into understanding the design process and building AI systems that can act as designers themselves.

2.1 Engaging Computers as Creative Equals

Recall that Lubart's four roles for computers in creativity support tools includes that of the "colleague", in which the computer collaborates with the user. However, looking at the ways in which humans interact with existing systems that embody this role, including Tanagra, reveals that the colleague is generally not considered an equal partner in the design process. The computer typically plays a subordinate role, obeying the instructions of the human designer, producing content to meet her goals, and never questioning her decisions unless there are logical contradictions. While this is undoubtedly a useful role for the computer to play, it is quite different from that of a human colleague.

Collaborative design between humans tends to be more a partnership of equals. For example, consider a scenario where two participants, one an engineer and the other an artist, work together to design a digital game. While each participant brings different skills to the table, they work together by brainstorming different ideas, asserting their expertise when appropriate, defining their own constraints, and negotiating towards a mutual goal. Furthermore, conversation between two designers is not strictly turn-based, unlike mixed-initiative systems where turn taking is the norm; the artist may interrupt the engineer to request clarifications, to correct assumptions or to suggest a new idea, or the two designers may work in parallel to create separate prototypes before combining their ideas. This kind of collaboration is essential in interdisciplinary fields, where there are several experts from different areas negotiating complex interdependent aspects of the project. There is great potential for the adoption of design support tools for novices in these fields, with the computer acting as a domain expert outside of the user's area of knowledge.

However, existing approaches to mixed-initiative design do not support this form of collaborative design with a computer. I suggest a refinement of this "colleague" role: the "computer as collaborator". In this capacity, the computer is elevated to equality with the human designer and is able to more strongly influence design decisions based on its expertise. There are three central issues in the creation of future "computer as collaborator" systems, issues that touch on both AI system design and user interface concerns. At the core of all of these issues is the need to imbue the AI system with an ability to argue about design: the system should understand and explain its reasons for any actions it takes, be

capable of engaging in a debate with the human about design, and compromise on design decisions.

2.1.1 Negotiating High-Level Design Goals

The collaborative design process is characterized by the negotiation of high-level design goals. Two participants may have drastically different ideas about the final product, be aware of different constraints, and have different approaches to solving problems. While these differences can offer some challenges, a diversity of opinions can also lead to more productive design conversations and a more interesting final product.

Current mixed-initiative systems have the human set all of the goals for the project, whether explicitly or implicitly, and the computer acts in fulfillment of these goals. Negroponte warns that computers in creative support roles should not push their own agenda on the human (Negroponte 2003), instead allowing the human to make all of the major design decisions and use the computer to amplify their own design potential. I agree that the computer should not dominate the design conversation; however, its expertise must be reflected when negotiating design goals, and both the suggestions it makes and the constraints it provides can lead to more productive design discussions. Note that this expertise could be more than what it is imbued with when it is created; human designers change and improve on their designs over time as they build experience. From its collaboration with a wide range of humans on a wide range of design projects, the AI could also be able to learn from its own past experiences.

2.1.2 Overriding Human Decisions

Current systems hold human decisions paramount over all other considerations in the system. For example, Songsmith's harmony generator (Simon et al. 2008) will never change the melody that has been written by the human, but can accommodate changes to the melody that are made later during composition. Human collaborators, in contrast, will frequently request that certain prior decisions be overridden in favor of new ideas. A game designer might suggest that a mechanic that was previously considered important be removed from the game to accommodate a change made to the game's story, for example. These changes are, ideally, done politely and with the cooperation of others on the design team.

An AI system that will collaborate as an equal with a human designer must therefore be able to override decisions made by the human as appropriate, and vice versa. This requires the AI system to record when decisions are made and the context in which they are made, so that it can infer which decisions are likely to be reversible and which are not. It also requires the AI system to be able to explain the decisions it is making in the context of larger design goals, so that the human understands what its collaborator is doing and why.

2.1.3 Moving Away from Turn-Taking

In order to support both design goal negotiation and decision overriding, the interface of a collaborative mixed-initiative system must support the computer interrupting the human designer. Current mixed-initiative systems are generally reactive, with a turn-taking "conversation" occurring between human and computer participants. However, conversation between human designers rarely follows a strict turn-taking model; there are

frequent interruptions when both parties have something to say on the topic at hand, and pauses when both parties must think about how to address a particular problem. Furthermore, neither participant is purely reactive to the other, as current mixed-initiative systems tend to be. Instead, each designer takes a proactive role in contributing to the final product.

To accommodate these changes, mixed-initiative interfaces must be modified to move away from the turn-taking model of interaction towards a more sophisticated design conversation. Two of Horvitz's design principles for mixed-initiative systems are "employing dialog to resolve key uncertainties" and "employing socially appropriate behaviors for agent-user interaction" (Horvitz 1999). These two principles are key in the move away from a turn-taking conversation; the computer must be able to politely interrupt the user with its own suggestions without annoying the user through needless dialogs.

2.2 Creating New Game Genres

Endless Web is an instance of a new game genre—the PCG-based game—that can only exist because of procedural content generation. There are few other PCG-based games in existence; however, there are plenty of other games that could be made that deeply incorporate PCG as *Endless Web* does. For example, one can imagine a diary-like game that would literally take a year to play, with generated content personalized for players based on knowledge about events in their life. Or a game in which the purpose is for players to influence a generator to create content that they can then share with friends. Tanagra was created as a design tool to assist designers, but it could also be used in a game itself, in

which there are certain goals placed in the physical space and the player must modify the level so that these goals can be reached by lemming-like autonomous players.

But while this dissertation has focused exclusively on the use of procedural content generation to enable new game genres, the future of games lies along many paths that don't all involve the incorporate of PCG. There are many other AI techniques, such as natural language generation, story planning, or player modeling, that can be used not only in the service of existing games but to enable their own new game genres. And there are also other, non-AI technologies, such as wearable computing, augmented reality, or e-textiles, that can be similarly used to create new kinds of games and attract a potentially broader audience than current digital games. There is a great deal of research to be done in how to incorporate these technologies into the game design process.

2.3 Tools for New Domains

The work presented in this dissertation has been focused on a particular kind of 2D platforming game. While there is clearly more work to be done in representing and generating more complicated aspects of these games—more complicated geometry types, the ability for the player to choose a path, and hidden areas are three examples—there is also a great amount that can be learned from applying the research done here to other domains, both in game design and other design domains.

2.3.1 Story and Level Design

I believe the beat timeline concept could extend well to other game genres where pacing is a salient concern during the design process. For example, first-person shooters are often

quite linear in terms of the path players take through levels, and the pacing of challenges is extremely important to a player's enjoyment of the game. However, there are many cases when pacing, while still a concern, is not the most important design concern. The role-playing game (RPG) is one such game genre. In these games, the story that the player follows and the choices the player makes when accepting, rejecting, and completing quests are among the most important elements of the game's design.

Level and quest design tools for these games are quite complex, due in large part to the nature of level design for 3D games, the need for triggers for NPC behavior, and the authoring burden associated with writing both the logic for quests and the dialog that instantiates them. Procedural content generation has already been used to help in the prototyping phase of level design, and there are tools to assist novices with authoring quests through the use of quest patterns (Cutumisu et al. 2007; Olivetti 2011). However, there is no tool available that brings the benefits of PCG and other AI techniques to bear on the combined problem of story and level design. It is clear that level design influences story and vice versa (Howard 2008; Smith et al. 2011d); being able to formalize the relationship between the two leads to the opportunity for tools that can assist both quest and level designers in the same way that Tanagra can assist level designers. In place of a beat timeline, one can imagine a quest graph; designers could rapidly see the effects of altering a quest's structure on the design of a level, or vice versa.

2.3.2 Other Design Domains

There is also the opportunity to use some of the techniques from this dissertation in domains outside of video games. There are plenty of other creative domains in which

designers can receive assistance from an artificial intelligence. This requires research in both formalizing aspects of that design domain such that a computer can understand it, and in the areas where users have difficulty expressing their intentions. For example, consider a PCG-assisted web design tool: a user specifies the information they wish to convey to readers (perhaps prompted by the tool) and some other simple facts about their website, and in response the tool generates possible websites for the user to select from and further manipulate. This dissertation has outlined only the beginning of this field of PCG-assisted, mixed-initiative design tools. There is a great deal of work yet to be done.

3 Conclusion

The technologies that have most heavily influenced game design thus far have been computer graphics and an increase in processing power, leading to a rise in increasingly immersive environments with mechanics primarily focused around movement and collision detection. Combat mechanics lend themselves well to what computers can understand best: mathematics. Players do not question the array of choices they have for how to engage in combat in games; complex strategies emerge from simple combat mechanics due to the ease with which computers can compute the collisions between objects and the mathematical equations governing combat rules. Furthermore, the creation of these games requires the efforts of skilled designers, engineers, and artists working together to build and understand the interactions between game rules, and design, implement, and decorate such immersive environments.

This dissertation has described two main systems—an intelligent tool to assist designers in creating content for 2D platforming games and a content generator that is sufficiently

expressive and controllable as to enable a new kind of game design—and a means for understanding their expressivity. The development of these systems is motivated by a desire to understand how procedural content generation can advance game design by taking advantage of the strengths of content generators, rather than attempting to force generators to entirely replace a human designer. More broadly, it is my hope that this work is an early step in broadening the subject matter that games can address. Sid Meier has famously said that "games are a series of interesting choices" (Johnson 2009); however, the domain in which these choices can be made is limited to those we can easily model. New games with new interesting choices can only be achieved by developing the technologies that can support them.

REFERENCES

.theprodukkt, 2004. *.kkrieger (PC Game)*, Available at: http://www.theprodukkt.com/kkrieger.

[adult swim games], 2010. *Robot Unicorn Attack (PC Game)*.

Adams, E., 2009. *Fundamentals of Game Design* 2nd ed., New Riders Press. Available at: http://www.amazon.com/Fundamentals-Game-Design-Ernest-Adams/dp/0321643372.

Allen, J.E., Guinn, C.I. & Horvtz, E., 1999. Mixed-initiative interaction. *Intelligent Systems and their Applications*, 14(5), pp.14–23.

Anon, Diorama Manual. Available at: http://warzone2100.org.uk/manual-diorama.html [Accessed April 30, 2012].

Anon, 2008. LittleBigPlanet's Little Big Computer. *SPOnG*. Available at: http://spong.com/article/16360/LittleBigPlanets-Little-Big-Computer [Accessed April 29, 2012].

Armor Games, 2008. *Shift (PC Game)*.

Artoon, 2006. *Yoshi's Island DS (Nintendo DS)*, Nintendo.

Ashmore, C. & Nitsche, M., 2007. The Quest in a Generated World. In *Proceedings of the 2007 Digital Games Research Association (DiGRA) Conference: Situated Play*. Tokyo, Japan, pp. 503–509.

Autodesk, 1990. *3DS Max (PC Software)*.

Autodesk, 1998. *Maya (PC Software)*.

Bennett, J., 1951. *Nimrod Computer*.

BioWare, 2009. *Dragon Age: Origins (PC Game)*, Electronic Arts.

Bjork, S. & Holopainen, J., 2004. *Patterns in Game Design (Game Development Series)* 1st ed., Charles River Media.

Bleszinski, C., 2000. The Art and Science of Level Design. Available at: http://www.cliffyb.com/art-sci-ld.html [Accessed January 15, 2011].

Blizzard North, 1997. *Diablo (PC Game)*, Blizzard Entertainment.

Booth, M., 2009. The AI Systems of Left 4 Dead. In *Keynote, Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE09)*. Palo Alto, CA.

Boutros, D., 2006. A Detailed Cross-Examination of Yesterday and Today's Best-Selling Platform Games. *Gamasutra*. Available at: http://www.gamasutra.com/view/feature/1851/a_detailed_crossexamination_of_.php [Accessed May 13, 2012].

Boyes, E., 2006. Q&A: David Braben -- from Elite to today. *GameSpot UK*. Available at: http://uk.gamespot.com/news/6162140.html?part=rss&tag=gs_news&subj=6162140 [Accessed December 27, 2011].

Braben, D. & Bell, I., 1984. *Elite (BBC Micro)*, Acornsoft.

Brathwaite, B. & Schreiber, I., 2008. *Challenges for Game Designers* 1st ed., Boston, MA: Charles River Media.

Byrne, E., 2004. *Game Level Design (Game Development Series)*, Charles River Media.

Carbonell, J.R., 1970. *Mixed-Initiative Man-Computer Instructional Dialogues*. PhD Thesis. Cambridge, MA: Massachusetts Institute of Technology.

Chen, S. et al., 2009. Evaluating the Authorial Leverage of Drama Management. In *Proceedings of the AAAI Spring Symposium on Intelligent Narrative Technologies II*. Palo Alto, CA.

Choco Team, 2008. Choco: an Open Source Java Constraint Programming Library. In *White Paper, 14th International Conference on Principles and Practice of Constraint Programming, CPAI08 Competition*. Sydney, Australia.

Co, P., 2006. *Level Design for Games: Creating Compelling Game Experiences*, New Riders Games. Available at: http://www.amazon.com/Level-Design-Games-Compelling-Experiences/dp/0321375971.

Compton, K. & Mateas, M., 2006. Procedural Level Design for Platform Games. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE06)*. Palo Alto, CA.

Cook, D., 2007. The Chemistry Of Game Design. *Gamasutra*. Available at: http://www.gamasutra.com/view/feature/129948/the_chemistry_of_game_design.php [Accessed May 1, 2012].

Cooper, S., Dann, W. & Pausch, R., 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5), pp.107–116.

Costikyan, G., 2006. I Have No Words and I Must Design. In K. Salen & E. Zimmerman, eds. *The Game Design Reader : A Rules of Play Anthology*. MIT Press.

Crane, D., 2011. GDC 2011 Panel: Pitfall Classic Postmortem With David Crane. *Gamespot*. Available at: http://www.gamespot.com/pitfall-the-mayan-adventure/videos/gdc-2011-panel-pitfall-classic-postmortem-with-david-crane-6302614/index.html [Accessed April 28, 2012].

Crane, D., 1982. *Pitfall! (Atari 2600)*, Activision.

Crawley, D., 2012. Six million LittleBigPlanet user levels created; here are some of the best. *Venture Beat*. Available at: http://venturebeat.com/2012/01/17/six-million-littlebigplanet-user-levels-created-here-are-some-of-the-best/ [Accessed April 29, 2012].

Cross, N., 2011. *Design Thinking: Understanding How Designers Think and Work*, Berg Publishers.

Csíkszentmihályi, M., 1991. *Flow: The Psychology of Optimal Experience*, HarperCollins.

Cutumisu, M. et al., 2007. ScriptEase: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1), pp.32–58.

Dahlskog, S. & Togelius, J., 2012. Patterns and Procedural Content Generation. In *Proceedings of the Workshop on Design Patterns in Games (DPG 2012), co-located with the Foundations of Digital Games 2012 conference*. Raleigh, NC.

Dart, I.M., De Rossi, G. & Togelius, J., 2011. SpeedRock: procedural rocks through grammars and evolution. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. Bordeaux, France: ACM.

Donovan, T., 2010. *Replay: The History of Video Games*, Yellow Ant.

Dormans, J., 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (co-located with FDG 2010)*. Monterey, CA.

Dormans, J., 2012. *Engineering Emergence: Applied Theory for Game Design*. PhD Thesis. Amsterdam: University of Amsterdam.

Dormans, J., 2005. The Art of Jumping. Available at: http://www.jorisdormans.nl/article.php?ref=artofjumping [Accessed April 28, 2012].

Douglas, A.S., 1952. *OXO (EDSAC computer game)*.

Draves, S., 1999. *Electric Sheep (PC Software)*.

Ebert, D.S., 2003. *Texturing & Modeling: A Procedural Approach*, Morgan Kaufmann.

Eladhari, M.P. et al., 2011. *AI-Based Game Design: Enabling New Playable Experiences*, Santa Cruz, CA: UC Santa Cruz Baskin School of Engineering.

Engelbart, D., 2003. Augmenting Human Intellect. In N. Wardrip-Fruin & N. Montfort, eds. *The New Media Reader*. Cambridge, MA: The MIT Press, pp. 93–108.

Epic Games, 2010. *Unreal Development Kit*,

Feil, J.H. & Scattergood, M., 2005. *Beginning Game Level Design*, Course Technology PTR.

Firaxis Games, 2005. *Civilization IV (PC Game)*, 2K Games.

Fischer, G., 1994. Domain-oriented design environments. *Automated Software Engineering*, 1(2), pp.177–203.

Fischer, G. et al., 1993. Embedding critics in design environments. *Knowledge Engineering Review*, 8, pp.285–285.

Frontier Developments, 2008. *LostWinds (Nintendo Wii)*, Square Enix.

Fry, B., 2008. *Visualizing Data: Exploring and Explaining Data with the Processing Environment*, O'Reilly Media.

Fullerton, T., 2008. *Game Design Workshop, Second Edition: A Playcentric Approach to Creating Innovative Games* 2nd ed., Morgan Kaufmann.

Gabler, K. & Carmel, R., 2008. *World of Goo (PC Game)*, 2D Boy.

Gaijin Games, 2011. *Bit.Trip Runner (PC Game)*, Aksys Games.

Gearbox Software & Feral Interactive, 2009. *Borderlands (XBox 360)*, 2K Games.

Gingold, C., 2003. *Miniature Gardens and Magic Crayons: Games, Spaces, & Worlds*. Master of Science in Information, Design & Technology. Georgia Institute of Technology. Available at: http://levitylab.com/cog/writing/thesis/.

Golding, J., 2010. *Building Blocks: Artist-Driven Procedural Buildings*, Game Developers Conference. Available at: http://gdcvault.com/play/1012655/Building-Blocks-Artist-Driven-Procedural.

Google, 2000. *SketchUp (PC Software)*.

Greenberg, A.C. & Woodhead, R., 1981. *Wizardry: Proving Grounds of the Mad Overlord (Apple II)*, Sir-tech Software, Inc.

Gygax, G., 1978. *Advanced Dungeons and Dragons: Players Handbook*, New York: TSR Hobbies.

Harmonix, 2007. *Rock Band (XBox 360)*, Electronic Arts.

Hastings, E.J., Guha, R.K. & Stanley, K.O., 2009. Automatic Content Generation in the Galactic Arms Race Video Game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4), pp.245–263.

Hastings, E.J. & Stanley, K.O., 2010. Interactive genetic engineering of evolved video game content. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games, co-located with the 2010 International Conference on the Fundations of Digital Games*. Monterey, CA.

Hendrikx, M. et al., 2011. Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications and Applications*.

Higinbotham, W., 1958. *Tennis for Two (Analog computer game)*, Brookhaven, NY: Brookhaven National Laboratory.

Horvitz, E., 1999. Principles of mixed-initiative user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '99. Pittsburgh, PA: ACM, pp. 159–166.

Howard, J., 2008. *Quests: Design, Theory, and History in Games and Narratives*, A K Peters Ltd.

Hullett, K. & Mateas, M., 2009. Scenario Generation for Emergency Rescue Training Games. In *Proceedings of the 2009 International Conference on the Foundations of Digital Games (FDG 2009)*. Orlando, FL.

Hullett, K. & Whitehead, J., 2010. Design Patterns in FPS Levels. In *Proceedings of the 2010 International Conference on the Foundations of Digital Games (FDG 2010)*. Monterey, CA.

Hunicke, R., 2005. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*. ACE '05. Portland, OR: ACM, pp. 429–433.

Hunicke, R., LeBlanc, M. & Zubek, R., 2004. MDA: A Formal Approach to Game Design and Game Research. In *Proceedings of the 2004 AAAI Workshop on Challenges in Game Artificial Intelligence*. AAAI Workshop on Challenges in Game Artificial Intelligence. San Jose, California: AAAI Press.

Interactive Data Visualization Inc., 2010. *SpeedTree (PC Software)*, Lexington, SC.

Isla, D., 2009. Next-Gen Content Creation for Next-Gen AI. In *Invited Talk: 2009 International Conference on the Foundations of Digital Games (FDG 2009)*. Orlando, FL.

Iyer, V. et al., 1997. A novel representation for rhythmic structure. In *Proceedings of the 23rd International Computer Music Conference*. Thessaloniki, Hellas, pp. 97–100.

Jennings-Teats, M., Smith, G. & Wardrip-Fruin, N., 2010. Polymorph: A Model for Dynamic Level Generation. In 2010 Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2010). Palo Alto, CA.

Johnson, S., 2009. Analysis: Sid Meier's Key Design Lessons. *Gamasutra*. Available at: http://www.gamasutra.com/view/news/23458/Analysis_Sid_Meiers_Key_Design_Lessons.php [Accessed May 9, 2012].

Juul, J., 2005. *Half-Real: Video Games between Real Rules and Fictional Worlds*, The MIT Press.

Kazemi, D., 2008. Metrics and Dynamic Difficulty in Ritual's SiN Episodes. *Orbus Gameworld*. Available at: http://orbusgameworks.com/blog/article/70/metrics-anddynamic-difficulty-in-rituals-sin-episodes-part-1 [Accessed April 29, 2012].

Kazemi, D., 2009. Spelunky's Procedural Space. *Tiny Subversions*. Available at: http://tinysubversions.com/2009/09/spelunkys-procedural-space/ [Accessed April 28, 2012].

Kelleher, C., Pausch, R. & Kiesler, S., 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. San Jose, CA, pp. 1455–1464.

Kline, D. & Hetu, L., 2011. AI of Darkspore (Invited Talk). *2011 Conference on Artificial Intelligence in Interactive Digital Entertainment (Palo Alto, CA)*. Available at: http://dankline.files.wordpress.com/2011/10/ai-in-darkspore-aiide-2011.pptx.

Koster, R., 2004. *A Theory of Fun for Game Design* 1st ed., Paraglyph Press.

Kremers, R., 2009. *Level Design: Concept, Theory, and Practice*, A K Peters/CRC Press.

Lager, C., 2009. Adam Atomic on Canabalt. Available at: http://www.gamingdaily.co.uk/2009/adam-atomic-on-canabalt/ [Accessed April 28, 2012].

Levenshtein, V.I., 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8), pp.707–710.

Licklider, J.C.R., 2003. Man-Computer Symbiosis. In N. Wardrip-Fruin & N. Montfort, eds. *The New Media Reader*. Cambridge, MA: The MIT Press, pp. 73–82.

Lubart, T., 2005. How can computers be partners in the creative process: Classification and commentary on the Special Issue. *International Journal of Human-Computer Studies*, 63(4-5), pp.365–369.

MacLaurin, M., 2009. Kodu: end-user programming and design for games. In *Proceedings of the 4th International Conference on Foundations of Digital Games*. FDG '09. Orlando, FL: ACM.

Manning, C.D., Raghavan, P. & Schütze, H., 2008. *Introduction to Information Retrieval*, Cambridge University Press.

Martin, A. et al., 2010. Evolving 3d buildings for the prototype video game subversion. In *Proceedings of the 2010 international conference on Applications of Evolutionary Computation*. EvoApplicatons'10. Barcelona, Spain: Springer-Verlag, pp. 111–120.

Mateas, M. & Stern, A., 2002. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4), pp.39–47.

Mateas, M. & Wardrip-Fruin, N., 2009. Defining operational logics. In *Proceedings of the Digital Games Research Association*. London, UK.

Mawhorter, P. & Mateas, M., 2010. Procedural Level Generation Using Occupancy-Regulated Extension. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG10)*. Copenhagen, Denmark, pp. 351–358.

Maxis, 2008a. *Spore (PC Game)*, Electronic Arts.

Maxis, 2008b. *Spore Creature Creator (PC Game)*, Electronic Arts.

Maxis, Sporepedia. Available at: http://www.spore.com/sporepedia [Accessed April 29, 2012].

Media Molecule, 2008. *Little Big Planet (Playstation 3)*, Sony Computer Entertainment.

Milam, D. & El Nasr, M.S., 2010. Analysis of Level Design "Push & Pull" within 21 Games. In *Proceedings of the 2010 International Conference on the Foundations of Digital Games (FDG 2010)*. Monterey, CA.

Montfort, N. & Bogost, I., 2009. *Racing the Beam: The Atari Video Computer System*, MIT Press.

Morch, A. & Girgensohn, A., 1991. JANUS: basic concepts and sample dialog. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '91. New Orleans, LA: ACM, pp. 457–458.

Müller, P. et al., 2006. Procedural Modeling of Buildings. *ACM Transactions on Graphics*, 25(3), pp.614–623.

Nagel, L. & Pederson, D.O., 1973. *SPICE: Simulation Program with Integrated Circuit Emphasis*, EECS Department, University of California, Berkeley.

NanaOn-Sha, 1999. *Vib Ribbon (PlayStation)*, Sony Computer Entertainment.

Negroponte, N., 2003. Soft Architecture Machines. In N. Wardrip-Fruin & N. Montfort, eds. *The New Media Reader*. Cambridge, MA, USA: The MIT Press, pp. 353–366.

Nelson, M., 2007. Breaking Down Breakout: System and Level Design for Breakout-style Games. *Gamasutra*. Available at: http://www.gamasutra.com/view/feature/130053/breaking_down_breakout_system_and_.php [Accessed April 16, 2012].

Nicollet, V., 2004. Difficulty in Dexterity-Based Platform Games. *GameDev.net*. Available at: http://www.gamedev.net/page/resources/_/creative/game-design/difficulty-in-dexterity-based-platform-games-r2055 [Accessed April 28, 2012].

Nintendo Creative Department, 1985. *Super Mario Bros. (NES)*, Nintendo.

Nintendo EAD, 2006a. *New Super Mario Bros. (Nintendo DS)*, Nintendo.

Nintendo EAD, 1990. *Super Mario World (SNES)*, Nintendo.

Nintendo EAD, 2006b. *The Legend of Zelda: Twilight Princess (Nintendo Wii)*, Nintendo.

Novak, J. & Castillo, T., 2008. *Game Development Essentials: Game Level Design*, Delmar Cengage Learning.

Novick, D.G. & Sutton, S., 1997. What is mixed-initiative interaction. In *Procedings of the AAAI Spring Symposium on Computational Models for Mixed Initiative Interaction*. pp. 114–116.

Nygren, N. et al., 2011. User-preference-based automated level generation for platform games. In *2011 IEEE Conference on Computational Intelligence and Games (CIG)*. pp. 55 –62.

Nyitray, K.J., 2011. William Alfred Higinbotham: Scientist, Activist, and Computer Game Pioneer. *Annals of the History of Computing, IEEE*, 33(2), pp.96 –101.

Ohkubo, H., 2003. *Warning Forever (PC Game)*, Hikware.

Olivetti, J., 2011. Storybricks: Opening the Pandora's box of MMO design. *Massively*. Available at: http://massively.joystiq.com/2011/08/08/storybricks-opening-the-pandoras-box-of-mmo-design/ [Accessed May 2, 2012].

Overmars, M., 1999. *GameMaker*, YoYo Games.

Perlin, K., 1985. An image synthesizer. In *ACM SIGGRAPH Computer Graphics*. pp. 287–296.

Persson, M., 2008. *Infinite Mario Bros! (PC Game)*, http://www.mojang.com/notch/mario/.

Persson, M., 2011. *Minecraft (PC Game)*, Available at: http://www.mojang.com/notch/mario/.

Procedural Inc., 2010. *CityEngine (PC Software)*, Zurich, Switzerland.

Pumpkin Studios, 1999. *Warzone 2100 (PC Game)*, Eidos Interactive. Available at: http://www.wz2100.net/.

Purho, P., 2009. *Crayon Physics Deluxe (PC Game)*.

Rareware, 1995. *Donkey Kong Country 2: Diddy's Kong Quest (Nintendo DS)*, Nintendo.

Regier, J. & Gresko, R., 2009. Random Asset Generation in Diablo 3. *Invited Talk, UC Santa Cruz*.

Resnick, M. et al., 2009. Scratch: programming for all. *Commun. ACM*, 52(11), pp.60–67.

Riedl, M. & O'Neill, B., 2009. Computer as Audience: A Strategy for Artificial Intelligence Support of Human Creativity. In Computational Creativity Support Workshop, co-located with CHI '09.

Robinson, A., 2009. Gearbox Interview: Randy Pitchford on Borderlands' 17 million guns. *ComputerAndVideoGames.com*. Available at: http://www.computerandvideogames.com/220328/interviews/gearbox-interview/ [Accessed April 14, 2012].

Rohrer, J., 2011. *Inside a Star-Filled Sky (PC Game)*.

Salen, K. & Zimmerman, E., 2004. *Rules of Play: Game Design Fundamentals*, The MIT Press.

Salge, C. et al., 2008. Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games. In *Proceedings of the 2008 ACM SIGGRAPH Sandbox Symposium on Video games*. Sandbox '08. Los Angeles, CA: ACM, pp. 7–14.

Saltsman, A., 2009. *Canabalt (PC Game)*, Adam Atomic. Available at: http://www.adamatomic.com/canabalt/.

Saltsman, A., 2010. Tuning Canabalt. *Gamasutra*. Available at: http://www.gamasutra.com/blogs/AdamSaltsman/20100929/6096/Tuning_Canabalt.php [Accessed April 28, 2012].

Satchell, C., 2009. Evolution of the Medium - Positioning for the Future of Gaming. In *Keynote: 2009 International Conference on the Foundations of Digital Games (FDG 2009)*. Orlando, FL.

Schell, J., 2008. *The Art of Game Design: A book of lenses* 1st ed., Morgan Kaufmann.

Secretan, J. et al., 2011. Picbreeder: a case study in collaborative evolutionary exploration of design space. *Evolutionary Computation*, 19(3), pp.373–403.

Shaker, N. et al., 2011. The 2010 Mario AI championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4), pp.332–347.

Shaker, N., Yannakakis, G.N. & Togelius, J., 2010. Towards Automatic Personalized Content Generation for Platform Games. In *Proceedings of the Sixth Artificial Intelligence in Interactive Digital Entertainment Conference (AIIDE10)*. Palo Alto, CA.

Shneiderman, B. et al., 2005. *Creativity Support Tools: a workshop sponsored by the National Science Foundation*, University of Maryland: National Science Foundation. Available at: http://www.cs.umd.edu/hcil/CST/Papers/creativitybook_final.pdf.

Simon, I., Morris, D. & Basu, S., 2008. MySong: automatic accompaniment generation for vocal melodies. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*. CHI '08. Florence, Italy: ACM, pp. 725–734.

Simutronics Corp. & Idea Fabrik Plc., 2011. *HeroEngine (PC Software)*.

Smelik, R. et al., 2011a. Semantic constraints for procedural generation of virtual worlds. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. PCGames '11. Bordeaux, France: ACM.

Smelik, R.M. et al., 2011b. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35(2), pp.352–363.

Smelik, R.M. et al., 2010. Integrating Procedural Generation and Manual Editing of Virtual Worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (co-located with FDG 2010)*. Monterey, CA.

Smith, A. et al., 2008a. Tableau Machine: A Creative Alien Presence. In *AAAI Spring Symposium on Creative Intelligent Systems 2008*. Palo Alto, CA.

Smith, A.M. et al., 2012a. A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game. In *Proceedings of the 2012 Conference on the Foundations of Digital Games*. Raleigh, NC.

Smith, A.M. et al., 2011a. An Inclusive View of Player Modeling. In *Proceedings of the 2011 International Conference on the Foundations of Digital Games (FDG 2011)*. Bordeaux, France.

Smith, A.M. & Mateas, M., 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3), pp.187 –200.

Smith, A.M., Nelson, M.J. & Mateas, M., 2009a. Computational support for play testing game sketches. In *Proceedings of the 5th Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. Palo Alto, CA, pp. 167–172.

Smith, A.M., Nelson, M.J. & Mateas, M., 2009b. Prototyping Games with BIPED. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE'09)*. Palo Alto, CA, pp. 14–16.

Smith, G. et al., 2011b. Launchpad: A Rhythm-Based Level Generator for 2D Platformers. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 3(1).

Smith, G. et al., 2012b. PCG-based Game Design: Creating Endless Web. In *Proceedings of the International Conference on the Foundations of Digital Games*. FDG '12. Raleigh, NC: ACM, pp. 188–195.

Smith, G. et al., 2011c. PCG-Based Game Design: Enabling New Playable Experiences through Procedural Content Generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, co-located with FDG 2011*. Bordeaux, France.

Smith, G. et al., 2009c. Rhythm-Based Level Generation for 2D Platformers. In 2009 International Conference on the Foundations of Digital Games (FDG 2009). Orlando, FL.

Smith, G. et al., 2011d. Situating Quests: Design Patterns for Quest and Level Design in Role-Playing Games. In 4th International Conference on Interactive Digital Storytelling (ICIDS 2011). Vancouver, BC, Canada.

Smith, G., Cha, M. & Whitehead, J., 2008b. A Framework for Analysis of 2D Platformer Levels. In *Proceedings of ACM SIGGRAPH Sandbox Symposium 2008*. Los Angeles, CA.

Smith, G., Whitehead, J. & Mateas, M., 2011e. Computers as Design Collaborators: Interacting with Mixed-Initiative Tools. In *Proceedings of the Workshop on Semi-Automated Creativity, co-located with ACM Creativity & Cognition 2011*. Atlanta, GA.

Smith, G., Whitehead, J. & Mateas, M., 2011f. Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on Computational*

Intelligence and AI in Games (TCIAIG), Special Issue on Procedural Content Generation, 3(3).

Sonic Team, 1991. *Sonic the Hedgehog (Genesis)*, SEGA.

Sonic Team, 1992. *Sonic the Hedgehog 2 (Genesis)*, SEGA.

Spinks, A., 2011. *Terraria (PC Game)*, Re-Logic.

Sternberg, R.J., 1999. Enhancing Creativity. In *Handbook of Creativity*. Cambridge University Press, pp. 401–402.

Sullivan, A. et al., 2011. Extending CRPGs as an Interactive Storytelling Form. In *Proceedings of the 2011 International Conference on Interactive Digital Storytelling*. Vancouver, BC, Canada, pp. 164–169.

Sullivan, A., Chen, S. & Mateas, M., 2009a. From Abstraction to Reality: Integrating Drama Management into a Playable Game Experience. In *Proc. of the AAAI 2009 Spring Symposium*.

Sullivan, A., Mateas, M. & Wardrip–Fruin, N., 2009b. Questbrowser: Making quests playable with computer-assisted design. In *Proceedings of the 8th Digital Art and Culture Conference*. Irvine, CA.

Sung, K., 2011. Recent Videogame Console Technologies. *Computer*, 44(2), pp.91–93.

Sutherland, I., 2003. Sketchpad: A Man-Machine Graphical Communication System. In N. Wardrip-Fruin & N. Montfort, eds. *The New Media Reader*. Cambridge, MA: The MIT Press.

Swink, S., 2008. Super Mario Brothers. In *Game Feel: A Game Designer's Guide to Virtual Sensation*. Morgan Kaufmann, pp. 201–228.

Takagi, H., 2001. Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. *Proceedings of the IEEE*, 89(9), pp.1275–1296.

Thue, D., Bulitko, V. & Spetch, M., 2008. PaSSAGE: A Demonstration of Player Modelling in Interactive Storytelling. In *Proceedings of the Fourth Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE '08)*. Palo Alto, CA: AAAI Press, pp. 227–228.

Togelius, J. et al., 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3), pp.172 –186.

Togelius, J., De Nardi, R. & Lucas, S.M., 2007. Towards automatic personalised content creation for racing games. In *IEEE Symposium on Computational Intelligence and Games 2007 (CIG07)*. Honolulu, HI, pp. 252–259.

Togelius, J., Preuss, M. & Yannakakis, G.N., 2010. Towards multiobjective procedural map generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (co-located with FDG 2010)*. Monterey, CA.

Toy, M. et al., 1980. *Rogue (PC Game)*.

Turtle Rock Studios, 2008. *Left 4 Dead (PC Game)*, Valve Corporation.

Tutenel, T. et al., 2009. Using Semantics to Improve the Design of Game Worlds. In *Proceedings of the Fifth Artificial Intelligence in Interactive Digital Entertainment Conference (AIIDE09)*. Palo Alto, CA.

Victor, B., 2012. Inventing on Principle. In *Keynote, Canadian University Software Engineering Conference 2012 (CUSEC 2012)*. Montreal, Quebec. Available at: http://vimeo.com/36579366.

Vona, M.A., 2009. *Virtual articulation and kinematic abstraction in robotics*. PhD. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

Wardrip-Fruin, N., 2009. *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*, The MIT Press.

Weber, B. et al., 2010. Reactive Planning Idioms for Multi-Scale Game AI. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG10)*. Copenhagen, Denmark.

Yu, D., 2009. *Spelunky (PC Game)*, Available at: http://www.spelunkyworld.com/.

Yujian, L. & Bo, L., 2007. A normalized Levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6), pp.1091–5.

Zimmerman, E., 2008. The Iterative Design Process. In *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann, pp. 16–19.