Synthesis Lectures on
Games and Computational Intelligence

SYNTHESIS
COLLECTION OF TECHNOLOGY

Matthew Guzdial · Sam Snodgrass ·
Adam J. Summerville

Procedural Content Generation via Machine Learning

An Overview

Springer

Synthesis Lectures on Games and Computational Intelligence

Series Editor

Daniel Ashlock, Guelph, ON, Canada

This series is an innovative resource consisting short books pertaining to digital games, including game playing and game solving algorithms; game design techniques; artificial and computational intelligence techniques for game design, play, and analysis; classical game theory in a digital environment, and automatic content generation for games.

Matthew Guzdial · Sam Snodgrass ·
Adam J. Summerville

Procedural Content Generation via Machine Learning

An Overview

Springer

Matthew Guzdial
University of Alberta
Edmonton, AB, Canada

Sam Snodgrass
Modl.ai
Copenhagen, Denmark

Adam J. Summerville
The Molasses Flood
Claremont, CA, USA

ISSN 2573-6485 ISSN 2573-6493 (electronic)
Synthesis Lectures on Games and Computational Intelligence
ISBN 978-3-031-16718-8 ISBN 978-3-031-16719-5 (eBook)
https://doi.org/10.1007/978-3-031-16719-5

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

In October 2016 at the very specifically named Embassy Suites by Hilton San Francisco Airport—Waterfront in Burlingame, California the three authors of this book stood around a cocktail table. We were all Ph.D. students at the time, and we had all come to realize we were working on roughly the same thing. Some of us called it learning-based approaches to PCG, some called it machine learned PCG, and some called it automated game design knowledge acquisition (a name nearly as wordy as the hotel's). However, we all understood that we'd hit on something new and exciting.

All of us have graduated since that conversation in that hotel. But, we all still find this area exciting and are actively working to push it forward. We wrote this book to share that excitement with you.

This book was a collective effort, not just from the three equally contributing authors. This book could not exist without the ongoing and vibrant PCG and PCGML community, both in and outside of academia. Thank you all for your passion, your scholarship, and your enthusiasm.

Edmonton, Canada — Matthew Guzdial
Copenhagen, Denmark — Sam Snodgrass
Claremont, USA — Adam J. Summerville
July 2022

Acknowledgments

Matthew would like to thank his fellow co-authors for agreeing to this immense undertaking, especially during the nightmare of the COVID pandemic. He'd also like to thank his husband, Jack, and the rest of their family for their endless enthusiasm and support. In particular, he'd like to thank his father, Mark, and his mother, Barbara, both rock star academics in Computer Science Education. You are a continual inspiration to him, and Matthew's glad to join you as published textbook authors. He would like to extend thanks to his students, colleagues, collaborators, and mentors. His life wouldn't be what it is without you all.

Sam would like to thank his wife, Jocelyn, for being incredibly supportive. From accommodating his strange hours when he was writing after work, to doing an enormous amount of labor to keep our day-to-day lives in order, and just generally making him happy in his life; Sam's contributions to this book would not have been possible without her. Thank you. He would also like to thank his family for their support and excitement around this project. And he would like to thank his colleagues, collaborators, and mentors who helped him get where he is. Of course he would also like to thank his cat, Sterling, for keeping him company during some weekend writing sessions. Lastly, he would like to thank his co-authors who, despite the time pressures and hellscapes, have been an absolute joy to write with.

Adam would like to thank his family for being so understanding and supportive during the journey to write this book. The throes of COVID have made finding the time, energy, and mental capacity to write this book extremely hard, and he knows this has not been easy for his children, Clark, Campbell, Maeve, and has been even harder for his wife, Mallorie, whose amazing work kept the family alive and afloat during the difficult authoring period. Mallorie, this book would not exist without you (or at least, Sam and Matthew would have had to write two more chapters each). He would also like to thank the mental health professionals who have helped him, specifically Laurie Ebbe Wheeler (he would also like to thank Vyvanse, without which editing would have been much harder). He would also like to thank his colleagues and mentors, including but not limited to, Michael Mateas, Noah Wardrip-Fruin, Ben Samuel, Joe Osborn, and James Ryan.

Finally, he would like to thank Matthew for subtweeting academic publishers and Sam for being a jovial, softening presence for his sometimes prickly co-authors.

We would all like to extend our heartfelt thanks to all reviewers and early readers of this book. We'd like to thank Dr. Jialin Liu and the other initial reviewer who chose to remain anonymous. In addition, we'd like to thank the 2022 members of Guzdial's GRAIL Lab for their detailed feedback and "book playtesting," particularly Adrian Gonzalez, Kristen Yu, Johor Jara Gonzalez, Anahita Doosti, Mrunal Sunil Jadhav, Akash Saravanan, Dagmar Lofts, Vardan Saini, Emily Halina, Kynan Sorochan, Jawdat Toume, Natalie Bombardieri, and Revanth Atmakuri. We'd especially like to acknowledge the use of game assets by Kenney.[1] Without his freely available game assets the figures in this book would look much worse.

The authors would like to dedicate this book to the memory of Dan Ashlock. Dan was a hugely influential figure in the academic games field. Dan believed in this book and went to bat for us with the publisher to get the book up and running. It is safe to say that this book would not have happened without his support, and we owe him a great thanks. Our field is diminished by his passing. We would also like to thank Joseph Brown for the introduction to Dan, and for serving as another instrumental force in making this book happen.

July 2022

Matthew Guzdial
Sam Snodgrass
Adam J. Summerville

[1] https://www.kenney.nl/assets

Contents

1 Introduction 1
- 1.1 Procedural Content Generation 2
- 1.2 Machine Learning 3
- 1.3 History of PCGML 4
- 1.4 Who is this Book For? 5
- 1.5 Who is this Book *Not* For? 5
- 1.6 Book Outline 6

2 Classical PCG 7
- 2.1 What is Content? 7
- 2.2 Constructive Approaches 8
  - 2.2.1 Noise 9
  - 2.2.2 Rules 11
  - 2.2.3 Grammars 12
- 2.3 Constraint-Based Approaches 13
- 2.4 Search-Based Approaches 16
  - 2.4.1 Evolutionary PCG 18
  - 2.4.2 Quality-Diversity PCG 20
- 2.5 Takeaways 21

3 An Introduction of ML Through PCG 23
- 3.1 Data and Hypothesis Space 24
- 3.2 Loss Criterion 27
- 3.3 Underfitting and Overfitting/Variance and Bias 31
- 3.4 Takeaways 33

4 PCGML Process Overview 35
- 4.1 Produce or Acquire Training Data 36
  - 4.1.1 Existing Training Data 37
  - 4.1.2 Producing Training Data 39
- 4.2 Train the Model 41
  - 4.2.1 Output Size 41

4.2.2 Representation Complexity . . . 42
4.2.3 Train, Validation, and Test Splits . . . 43
4.3 Generate Content . . . 45
4.3.1 Exploration vs. Exploitation in Generation . . . 45
4.3.2 Postprocessing . . . 46
4.4 Evaluate the Output . . . 47
4.5 Takeaways . . . 49

5 Constraint-Based PCGML Approaches . . . 51
5.1 Learning Platformer Level Constraints . . . 51
5.2 Learning Quest Constraints . . . 56
5.3 WaveFunctionCollapse . . . 60
5.3.1 Extract . . . 60
5.3.2 Observe . . . 63
5.3.3 Propagate . . . 64
5.3.4 Extending WaveFunctionCollapse . . . 65
5.4 Takeaways . . . 65

6 Probabilistic PCGML Approaches . . . 67
6.1 What are Probabilities? . . . 67
6.1.1 Learning Platformer Level Probabilities . . . 69
6.2 What are Conditional Probabilities? . . . 72
6.2.1 Learning Platformer Level Conditional Probabilities . . . 74
6.3 Markov Models . . . 78
6.3.1 Markov Chains . . . 78
6.3.2 Multi-dimensional Markov Chains . . . 79
6.3.3 Markov Random Fields . . . 81
6.3.4 Other Markov Models . . . 85
6.4 Bayesian Networks . . . 85
6.5 Latent Variables . . . 87
6.5.1 Clustering . . . 88
6.6 Takeaways . . . 90

7 Neural Networks—Introduction . . . 91
7.1 Stochastic Gradient Descent . . . 95
7.2 Activation Functions . . . 98
7.3 Artificial Neural Networks . . . 101
7.4 Case Study: NN 2D Markov Chain . . . 105
7.5 Case Study: NN 1D Regression Markov Chain . . . 107
7.6 Case Study: NN 2D AutoEncoder . . . 109
7.7 Takeaways . . . 112

8 Sequence-Based DNN PCGML 113
8.1 Recurrent Neural Networks 115
8.2 Gated Recurrent Unit and Long Short-Term Memory RNNs 117
8.2.1 Long Short-Term Memory RNNs 118
8.3 Sequence-Based Case Study—Card Generation 120
8.4 Sequence-to-Sequence Recurrent Neural Networks 122
8.5 Transformer Models 127
8.5.1 Case Study—Sequence to Sequence Transformer for Card Generation 130
8.6 Practical Considerations 132
8.7 Takeaways 133

9 Grid-Based DNN PCGML 135
9.1 Convolutions 135
9.2 Padding and Stride Behavior 142
9.3 Generative Adversarial Networks 148
9.4 Practical Considerations 153
9.5 Case Study—CNN Variational Autoencoder for Level Generation 154
9.6 Case Study—GANs for Sprite Generation 156
9.7 Takeaways 158

10 Reinforcement Learning PCG 159
10.1 One-Armed Bandits 160
10.2 Pixel Art Example 162
10.3 Markov Decision Process (MDP) 164
10.4 MDP Example 166
10.5 Tabular Q-Learning 169
10.5.1 Rollout Example 170
10.5.2 Q-Update 171
10.5.3 Q-Update Example 172
10.5.4 Rollout Example 2 173
10.5.5 Tabular Q-learning Wrap-up 174
10.6 Deep Q-Learning 174
10.7 Application Examples 176
10.8 Takeaways 178

11 Mixed-Initiative PCGML 181
11.1 Existing PCG Tools in the Wild 182
11.1.1 Classical PCG Tools 182
11.1.2 Microsoft FlightSim 183
11.1.3 Puzzle-Maker 184
11.2 Structuring the Interaction 185
11.2.1 Integrating with the PCGML Pipeline 186

11.2.2 Understanding the Model . . . 188
11.2.3 Understanding the User . . . 192
11.3 Design Axes . . . 194
11.3.1 AI vs. User Autonomy . . . 194
11.3.2 Static vs. Dynamic Model Systems . . . 196
11.4 Takeaways . . . 198

12 Open Problems . . . 201
12.1 Identifying Open Problems . . . 202
12.2 Problem Formulation . . . 203
12.2.1 Underexplored Content Types . . . 203
12.2.2 Novel Content Generation . . . 205
12.2.3 Controllability . . . 207
12.3 Input . . . 208
12.3.1 Data Sources . . . 208
12.3.2 Representations . . . 209
12.3.3 Data Augmentation . . . 210
12.4 Models and Training . . . 211
12.5 Output . . . 211
12.5.1 Applications . . . 211
12.5.2 Evaluation . . . 213
12.6 Discussion . . . 213

13 Resources and Conclusions . . . 215
13.1 PCGML Resources . . . 215
13.1.1 Other Textbooks . . . 216
13.1.2 Code Repositories . . . 217
13.1.3 Libraries . . . 218
13.1.4 Datasets . . . 219
13.1.5 Competitions and Jams . . . 220
13.1.6 Venues . . . 220
13.1.7 Social Media . . . 223
13.2 Conclusions . . . 224

References . . . 225

About the Authors

Matthew Guzdial is an Assistant Professor in the Computing Science Department of the University of Alberta and a Canada CIFAR AI Chair at the Alberta Machine Intelligence Institute (Amii). His research focuses on the intersection of machine learning, creativity, and human-centered computing. He is a recipient of an Early Career Researcher Award from NSERC, a Unity Graduate Fellowship, and two best conference paper awards from the International Conference on Computational Creativity. His work has been featured in the BBC, WIRED, Popular Science, and Time.

Sam Snodgrass is an AI Researcher at Modl.ai, a game AI company focused on bringing state-of-the-art game AI research from academia to the games industry. His research focuses on making PCGML more accessible to non-ML experts. This work includes making PCGML systems more adaptable and self-reliant, reducing the authorial burden of creating training data through domain blending, and building tools that allow for easier interactions with the underlying PCGML systems and their outputs. Through his work at Modl.ai he has deployed several mixed-initiative PCGML tools into game studios to assist with level design and creation.

Adam J. Summerville is the lead AI Engineer for Procedural Content Generation at The Molasses Flood, a CD Projekt studio. Prior to this, he was an Assistant Professor at California State Polytechnic University, Pomona. His research focuses on the intersection of artificial intelligence in games with a high-level goal of enabling experiences that would not be possible without artificial intelligence. This research ranges from procedural generation of levels, social simulation for games, and the use of natural language processing for gameplay. His work has been shown at the SF MoMA, SlamDance, and won the audience choice award at IndieCade.

Introduction

1

This book focuses on **Procedural Content Generation via Machine Learning** (**PCGML**), the generation of media or content with machine learning techniques [195]. While machine learning (ML) has been used to generate a wide variety of content including visual art, music, and stories, PCGML (and this book) focus largely on video games. Therefore, we focus on the types of content specific to video games, such as levels, mechanics, game character art, sound effects, game narrative, and so forth. Video games are a difficult medium for ML, due to their complexity and lack of training data as we'll discuss further below. But this difficulty is part of what makes the problems in this field so compelling.

PCGML takes many different forms depending on the kind of content we wish to generate and the ML technique we apply. But for the purposes of an introduction you can imagine taking some amount of game content (levels, mechanics, etc.), training an ML model on this data, and then using the trained ML model to generate more content similar to the training data. This basic, intuitive process approximates many of the techniques discussed in this book. However, it also demonstrates some core problems with PCGML. How do we generate content that is not just a small variation on what we already have? How can we ensure the output has the desired characteristics of the input? What do we do if we don't have any data, or only a small amount? And so on.

In this chapter, we'll introduce the very basic concepts of PCGML in terms of its two component parts: Procedural Content Generation (PCG) and Machine Learning (ML), along with a discussion of the relationship between the two. We'll then discuss a brief history of PCGML as a means of situating this book in this constantly evolving area, and identify our intended audience for this book. We'll end by briefly outlining the rest of this book, and some various ways to read it. Readers with some expertise in PCGML or a related area may wish to start with the final section of this chapter.

M. Guzdial et al., *Procedural Content Generation via Machine Learning*,
Synthesis Lectures on Games and Computational Intelligence,
https://doi.org/10.1007/978-3-031-16719-5_1

1.1 Procedural Content Generation

Procedural Content Generation (PCG) refers to the algorithmic generation of game content. By game content, we mean the various component parts of a game, including scripts or game code, levels or maps, character sprites or 3D models, animations, music, sound effects, and so on. By algorithmic generation we indicate some process or set of rules rather than typical human creation. Most often "algorithmic" indicates the use of computer code, and that's the way we'll discuss it in this book, but it could involve any set process or rules, such as generation via cards or dice [55]. PCG can look like the landscapes of Minecraft, the conversations and dungeons of Hades, or almost everything present in No Man's Sky.

PCG in the video game industry is most commonly used during development time, when it is used at all. That means that most PCG is invisible to the players of games. For example, most modern, open-world games make use of some amount of PCG to create their worlds. For example, the developers of the often-re-released Skyrim generated the game world's landscape with PCG. From there, human developers went back through the landscape, tweaked it, and added extra content like decorations, 3D models, enemies, quests, and so on. PCG is sometimes visible to players, particularly when it shows up inside games "at runtime," as in the examples of Minecraft, Hades, and No Man's Sky, though this is less common.

The examples we have given thus far are examples of classical PCG. Classical PCG, which we will discuss further in Chap. 2, relies on classical Artificial Intelligence (AI) methods like grammars and search [160]. Classical PCG has seen some adoption by the video game industry, and even in other, related industries. For example, Speedtree, a library that allows designers to quickly generate many unique 3D tree models with PCG, has been used in many AAA games, and even in Hollywood films like the Avengers series. However, most game content is made without PCG. In fact, PCG sometimes sinks projects, such as in the case of Mass Effect: Andromeda, a follow-up to the Mass Effect franchise which lost nearly two years of development time to an attempt to create a procedural galaxy in which the game would take place. The issue here is that working with PCG is tricky, it takes specialized design and algorithmic knowledge, along with buy-in from the entire development team. Therefore, it can often take more time and resources than just developing the game using more standard industry practices.

This design and algorithmic knowledge is the core requirement in creating a PCG generator. Specifically, PCG requires that the user hand-author knowledge to construct their generator so that it can output the kinds of content the user wants, and not the kinds they don't. In a grammar, a developer might need to author chunks of content and rules for how they fit together. You can think of this like making individual Lego bricks, which can then be used to create many different structures. In a search-based approach, a developer instead needs to author a representation of the search space (a space where every point is a piece of content), neighbor functions defining how to move through that space, and a fitness function to indicate what high quality content looks like in that space. More on both of these

types of approaches in Chap. 2. Tweaking this knowledge is key to shaping the output of the algorithm, and is an art in and of itself. As Kate Compton famously put it, it's difficult to solve the "10,000 bowls of oatmeal problem" [26]. It's easy to generate a lot of *something*, but it's tricky to make that something *interesting*.

Intuitively, we might consider learning this knowledge instead of having to hand-author it. After all, there's a large number of existing, high quality games. If we could learn to design based on the content from these existing games, we might be able to empower more people to benefit from PCG. This would allow more people to make games, and even for the creation of new types of games and experiences that would be impossible or impractical with modern game development practices. For example, consider how modern "open world" games are still limited to a single region, how games could adapt to their players, or even how players could create their own content for the game as they play. The list goes on, and we'll discuss the future potential for PCGML further in Chap. 12.

1.2 Machine Learning

Machine Learning (ML) refers to algorithms that "learn" the values of variables from data or experience. While it's an AI approach that can lead to amazing things, there has been a great deal of misinformation spread about machine learning. Essentially, it is just a way of adapting a function based on data. As a simple example, let's say we have the function $w * x + b = y$. This function takes in an x as an input argument, which should be a number, and outputs another number y. There are two variables in this function: w and b. Depending on the values of w and b we'll get different outputs for different x inputs. If you haven't recognized it already, this is just the function to describe a line. If we have enough examples of x inputs and their associated y outputs, we can approximate what the best values of w and b would be to match these. We're approximating a function ($w * x + b = y$) to match some training data (our pairs of x's and y's). You can see a visualization of this, with slightly different variable names, in Fig. 3.4. While there's lots of different kinds of machine learning, and at times the functions can get pretty complicated (to the point where we forget they're just functions and we start calling them models), these same basic principles stand. We'll focus on machine learning from Chap. 3 onward.

Modern ML approaches tend to struggle on a number of types of problems: (1) problems with low amounts of training data, (2) problems where the data has high **variance** (a lot of differences between pieces of data), and (3) problems without clear metrics for success. PCGML includes all three of these types of problems. There is typically a very small amount of training data available for a particular kind of game content, in comparison to datasets of non-game content. ImageNet, a common image classification dataset for machine learning, contains roughly 1.3 million images. In comparison, there are estimated to only be about 1.2 million published video games available for purchase [185]. Compared to images, video games are much more complicated and differ much more from one another. Because of

that, we'd actually expect to need significantly more data to model games compared to images. Further, across these 1.2 million published video games there's no consensus about what makes a good game. This is a positive thing, as different games are better suited to different people. However, this does mean there are no clear metrics for things like how fun a particular type of game content might be. This means that, in most cases, we can't just optimize for game quality.

The problems that PCGML confronts us with aren't just limited to games. In fact, anytime we want to model the output of individual humans these problems arise, since individuals can only produce so much data, humans differ from one another, and there's limited metrics for replicating human evaluation. Thus, solving these problems for PCGML can allow us to push the boundaries of what is possible with ML broadly.

1.3 History of PCGML

Procedural Content Generation via Machine Learning (PCGML) is the algorithmic generation of game content using machine learning methods. It was proposed to try to address problems in PCG and ML, and has enjoyed a great deal of popularity since. However, it's still a very young field, and we'll try to reflect that in this section and throughout this book.

In 2013, Sam Snodgrass and Santiago Ontañón published "Generating Maps using Markov Chains" the first paper later recognized as an example of PCGML [176]. In it, they discussed a project training a Markov Chain (a type of probability-based ML model we'll discuss more in Chap. 6) on Super Mario Bros. levels in order to generate new Mario levels. In this paper, the authors simply referred to their approach as a "learning-based approach to PCG." Other level generators had used machine learning as part of the generation process, such as the level generator of Robin Baumgarten from the 2010 Mario AI Championship Level Generation track, but this generator still relied on hand-authored chunks of levels, and sequenced these hand-authored chunks based on an ML analysis of player behavior [157]. Similarly, the Ludi system took in existing game content (in this case whole board games) as input in order to generate new board games, but no learning occurred [20]. Instead, the system recombined the existing games without altering anything about the generation approach based on the input. That's why we point to Sam and Santi's 2013 paper as the beginning of PCGML.

From 2014 to 2016, a large number of early PCGML systems debuted. A number of additional Markov Chains methods were published [33, 180]. Researchers began to focus on the problem of acquiring sufficient training data [59, 200], and the first neural network PCG experiments were published [77, 198]. At this point, the authors of this book and a large cadre of other early PCGML researchers began work on a survey paper of this growing area. Together, we would dub it Procedural Content Generation via Machine Learning [195].

Since the survey paper, PCGML has continued to grow as a research area. WaveFunctionCollapse, a simple and low-data PCGML approach, began to gain popularity as an approach

among indie game developers before being picked up by academic researchers [88]. There were the first attempts at generating entirely novel games with PCGML [63, 150], and at trying to create PCGML tools for designers [35, 57, 154]. But overall, the fundamental problems of PCGML remain unsolved, including the problems discussed above and many more remaining. We will discuss some of these open problems in Chap. 12.

1.4 Who is this Book For?

Our hope is that this book is accessible to PCG practitioners, ML practitioners, and anyone interested in these topics. The book can be used as the basis for a class, with every chapter serving as the basis of 1-2 lectures, as an introduction to these topics, or simply as a reference or guide. Our hope is that this book can demystify ML for those on the game design and PCG side of things, and make the benefits of applying ML to PCG clear for those on the ML side of things. For a class, we have written this book to be programming language agnostic, but it will require at least some understanding of coding. We recommend using this book for students at least at the undergraudate level, as many of the concepts in the book rely on fairly complex mathematics, though we'll do our best to express these clearly. Our hope is that the individual chapters can serve as a reference and guide for individuals looking to implement particular PCGML approaches, or for those interested in conducting PCGML research.

1.5 Who is this Book *Not* For?

While it may seem natural to some readers, we won't be covering reinforcement learning for automated game playing (the technology behind AlphaGo, OpenAI Five, AlphaStar, etc.) in this book. We will cover how reinforcement learning can be applied to PCG in Chap. 10, but not how to train agents to play or interact with existing content. There are many excellent resources on this subject, but this is not one of them. We focus on game design problems, not game playing problems.

We also do not intend this book to be an all encompassing look at Procedural Content Generation or Machine Learning as individual fields. We instead focus on the intersection of these two fields. While we'll introduce concepts from both as they are relevant to PCGML, if you find you want to dig deeper we recommend seeking out introductory texts on PCG [160] and/or ML [125].

1.6 Book Outline

In this section we'll briefly discuss the chapters of the book and some suggested reading orders depending on your level of familiarity. If you are already familiar with PCG, you can safely skip Chap. 2. Similarly, if you are already familiar with ML, you can safely skip Chap. 3. We recommend reading Chap. 4 regardless of your familiarity with these topics, as we overview the PCGML project process we'll use in this book. From there, readers can skip around as they like depending on their level of familiarity with the chapter topics. However, newcomers would likely benefit from reading the chapters in order. We recommend ending with Chaps. 12 and 13, regardless of your reading order. The chapters will cover the following topics:

- **Chapter** 2 presents an overview of "classic" approaches to PCG, which do not make use of machine learning.
- **Chapter** 3 covers the basic concepts necessary to understand the machine learning aspects of this book.
- **Chapter** 4 overviews our process for PCGML projects, along with covering practical and ethical considerations.
- **Chapter** 5 focuses on our first PCGML area: ML constraint-based approaches. This chapter covers the most commonly applied PCGML approaches in industry at the time of writing.
- **Chapter** 6 covers our second PCGML area: probabilistic models. These are some of the simpler PCGML approaches, particularly for those with a background in probability.
- **Chapter** 7 begins our coverage of deep neural networks (DNNs) for PCGML, starting with the basics. We recommend reading this chapter before Chaps. 8 and/or 9.
- **Chapter** 8 covers DNN models for processing sequences like text that can be applied to PCGML.
- **Chapter** 9 covers DNN models for processing image-like data structures that can be applied to PCGML.
- **Chapter** 10 focuses on our last major PCGML area: PCG via Reinforcement Learning or PCGRL. This differs significantly from the rest of this book due to not relying on existing training data.
- **Chapter** 11 introduces mixed-initiative PCGML, incorporating PCGML into tools for designers. This is an open problem but with existing applications, which we cover in this chapter.
- **Chapter** 12 overviews other open problems in PCGML (at the time of writing). These topics could serve as the basis for a research project or thesis.
- **Chapter** 13 ends with our conclusions, some discussion, and a variety of resources for PCGML practitioners.

Classical PCG

2

The other chapters in this book cover a wide range of approaches to procedural content generation that leverage different machine learning paradigms. Before jumping into the machine learning-based approaches, we will use this chapter to give a brief introduction to classical (i.e., non-machine learning-based) PCG approaches to provide context for the remainder of the book. In particular, we will introduce and discuss **constructive**, **constraint-based**, and **search-based** PCG as PCG paradigms that do not rely on machine learning. Constructive PCG relies on hand-authored rules and functions for assembling new pieces of content. Constraint-based PCG approaches define what a "valid" piece of content is using constraints, and use those constraints to find new content. Finally, search-based PCG defines the space of content, and uses optimization procedures to find high quality content within that space.

Each of the groups outlined above use unique methods for generating content. For each of these groups we will highlight the input needed from the user or developer, how that approach works at a high level, and examples of how related approaches have been or can be used. Lastly, we will discuss possible connections and extensions to PCGML. However, before we begin discussing these paradigms, we will give a brief overview of types of content we might want to generate.

2.1 What is Content?

When hearing about procedural content generation, you may think "What do they mean by content?" or "What can we generate?" The idealistic answer is that pretty much any part of a game (be it structural or mechanical) can be considered **content** or something that we could try to generate. Everything from game levels to textures to stories to gameplay mechanics to full games have been procedurally generated. A full categorization of content types is

M. Guzdial et al., *Procedural Content Generation via Machine Learning*,
Synthesis Lectures on Games and Computational Intelligence,
https://doi.org/10.1007/978-3-031-16719-5_2

outside of the scope of this book, but there are existing discussions around types of content. Hendrikx et al. [72] categorize game content hierarchically, starting at the bottom with *game bits* (i.e., the atomic game elements), then up to *game spaces* (i.e., the environment or world where the player and agents interact with the game), all the way up to *game designs* and through to *derived content* (i.e., content or information derived from the game, such as leaderboards). The detailed description provided by Hendrikx et al. makes this paper a good resource if you want a deeper discussion of content types. In the original PCGML survey paper [195] we (along with the other authors) instead focused on the representation of the content. We grouped content (regardless of its function) according to its structure: *sequence*, *grid*, or *graph*. This categorization could be useful when considering how to represent your chosen content type, and what implications that might have on the appropriate approach. Liu et al. [116] use a flat structure of content types: *game levels*, *text*, *character models*, *textures*, and *music and sound*. They give an overview of how different machine learning and especially deep learning methods have been applied across these categories of content; as such, it is a good resource if trying to decide on an approach or model architecture to use for a certain type of content. Notice, however, that the same content can be represented in many different ways, each of which lends itself to certain ML techniques. Similarly, disparate types of content might be represented in the same way, leading to similar ML applications. For instance, a level might be represented as an image (like textures), it might be represented as a sequence (like text), or it might be represented as a collection of content oriented in space (like a character model). As such, in this book we will tend towards representational categories (e.g., sequences, grids, graphs).

Categorizing content types can be a useful tool or lens through which to view PCG. But more importantly, while reading this book try to keep an imaginative mind. When we introduce a new approach we will give examples of how it has been used, and perhaps how it could be used in the future, in order to provide context and hopefully deeper understanding of the technique. But as you read try to also think of new ways the approach could be leveraged (e.g., new content types, new representations, new applications). PCGML is a young field, and there is a lot of unexplored space; so keeping an inquisitive eye open as you become acquainted with the field could lead to the next big innovation!

2.2 Constructive Approaches

Constructive procedural content generation describes the family of approaches that quickly generate a piece of content using rules and randomness, often in a one-shot fashion [158, 209]. Constructive approaches rely on the encoded design and domain knowledge from the creator of the approach. The approach then directly uses this encoded knowledge to create new content. Since these approaches directly rely on the encoded domain knowledge of the user, they allow the creator a lot of control over the generative process. Additionally, these approaches tend to be somewhat simple (in the algorithmic and conceptual space), making

them more accessible and as such the most common family of approaches used in games. An example of a conceptually simple approach, is one where the designer creates two segments for a level, and a rule that says "flip a coin to decide which section should be placed next." The encoded knowledge in this situation is the hand-authored level segments as well as the rule for how to place them. Now, this toy example serves two purposes: to give some insight into how a simple constructive PCG system might work, and to highlight what is needed from a human designer of such a system. An important concept to remember here is that constructive PCG systems are often one-shot generators (i.e., will create a piece of content without relying on feedback or being expected to further improve on that generated piece). This has the implication that either (1) the encoded knowledge, rules, etc. need to ensure that "bad" content cannot be created, (2) the generator needs to be used in scenarios where low quality content is permissible, or (3) there is a human at the end of the system choosing the "good" outputs.

The toy example above is not representative of all constructive PCG and the encoded knowledge used in constructive PCG can take many forms. In the following subsections we will discuss constructive approaches that leverage an increasing amount of hand-authored encoded knowledge starting with **noise** (structured randomness), moving to **rules** (directly encoded knowledge), and closing with **grammars** (more formally structured rule encodings).

2.2.1 Noise

Noise refers to a family of structured random functions. In graphics and PCG noise functions can be described as functions that give random real values[1] in an interval (commonly, 0 and 1) over some domain (i.e., 1-dimensional, 2-dimensional, etc.). White noise is one of the most commonly known types of noise functions, and can be thought of as sequences of independent random variables over an interval (e.g., a grid with each value randomly chosen between 0 and 1) with constant density across all frequencies. In addition to white noise, there are other types of noise with various properties. For example, while white noise is uniformly distributed over frequency ranges (i.e., will have quickly changing values and slowly changing values with equal likelihood), pink noise has a denser distribution around the lower frequency ranges (i.e., slowly changing values are more likely than quickly changing values). Alternatively, Blue noise has a denser distribution around higher frequency ranges (i.e., quickly changing values are more likely than slowly changing values). If thinking of images, pink noise will tend to have smoother transitions between colors, such as going from a dark to light grey spread over many pixels; blue noise will tend to have transitions from light and dark more quickly in only 1 or a few pixels; and white noise will have both

[1] Here we mean *real* in the mathematical sense (i.e., a value in $\mathbb{R}$). That is, essentially a continuous number that can be written with infinite decimal point precision.

of these patterns occurring. A detailed survey by Lagae et al. [101] covers different types of noise as used in image processing and graphics.

In the context of constructive PCG, the knowledge being encoded by the designer when using a noise-based approach, is which noise function has the properties most useful for the domain and what the sampled noise represents in that domain. Blue noise, because of how it is sampled and distributed, can be useful for pseudo-randomly distributing objects somewhat evenly throughout a space or level [2], whereas pink noise might be better suited to generating landscapes with more slowly changing height values. Below we give a brief introduction to a few noise-based approaches to texture and terrain modeling, but the reader is referred to the book by Ebert et al. [41] for a more detailed look at these topics.

Texture and Terrain Synthesis

Noise functions are commonly used in the creation of procedural textures and terrain (or heightmaps). Specifically, generating natural textures such as wood, marble, and clouds often rely on the randomness and detail provided by noise functions [101]. Examples of this can be found as far back as the 1980s in the graphics community where applying noise to textures led to more realistic looking natural textures [136]. Since then, there has been much more work in procedural textures and image synthesis using noise functions [6, 80, 102], as well as in terrain generation [5, 168].

A common extension to noise-based approaches leverages a hierarchical (or multi-resolution) strategy. In these multi-resolution approaches, noise is sampled at various resolutions and combined together to form the final result. For example, we can use a white noise function to sample values between 0 and 1 on a 16×16 grid (Fig. 2.1 first). Next, we use the same noise function to sample values between 0 and 0.5 on a larger 32×32 grid (Fig. 2.1 second), 0 and 0.25 on a larger 64×64 grid (Fig. 2.1 third), and finally 0 and 0.125 on the largest 128×128 grid (Fig. 2.1 fourth). In each of these steps, we double the height and width of the grid and halve the range of the noise values; this forces the approach to create different scales of features (i.e., bigger features in the initial grid, and smaller details in the

Fig. 2.1 This figure shows sampled noise at different resolutions and magnitude ranges. The left-most grid was sampled at a 16×16 resolution with a range of 0 to 1, with the values between filled in with bicubic interpolation. The second grid was sampled at a resolution of 32×32 with a range of 0 to 0.5, and interpolated in the same way. The third grid was sampled at a 64×64 resolution with a range of 0 to 0.25, and the fourth grid was sampled at a resolution of 128×128 with a range of 0 to 0.125. The final grid is the result of averaging the grids together, which could then be used for a terrain heightmap or a texture

larger grids). We then scale up the lower resolution grids (i.e., the 16×16, the 32×32, and 64×64 grids) to the full size (i.e., the 128×128) by interpolating the values between the sampled points. Lastly, we combine the values at each position together, which gives us a texture or terrain with features at different levels of granularity (Fig. 2.1 last). In our case, we averaged the values together, but adding and normalizing is also common. This final grid can be used as a heightmap for terrain where brighter pixels are higher altitudes and the darker pixels are lower altitudes. We could even translate the darkest pixels to water or being below sea level.

2.2.2 Rules

Rule-based PCG techniques are a subset of constructive approaches that create content by leveraging a set of manually-defined rules, and often rely on either designer-created chunks of content or designer-created templates. Rule-based PCG systems are among the most commonly used PCG approaches in commercial games due to the high amount of control and designer input they are able leverage.

For example, we can imagine a scenario where a designer has created a set of different rooms that a player might interact with and investigate. Some of these rooms might require the player's character to fight enemies, some might have treasure chests, some might allow the player's character to purchase items, and some might have special events that get triggered when the player enters. The designer doesn't want the players to experience the same sequence of rooms every time, and so instead of placing the rooms in a set layout, they devise some rules for how the rooms can be laid out (e.g., only place 1 treasure room in the map, don't place more than 5 combat rooms near each other, don't place extra strong enemies too early in the map, only allow certain types of events under different scenarios, etc.) This rule-based constructive PCG approach relies on the designer encoding their knowledge and desires for the game into smaller content blocks (rooms) as well as into the rules for how to place the content blocks. This is a fairly common approach in rule-based PCG approaches, and in fact, the deckbuilding game *Slay the Spire*, and the dungeon crawling game *Hades* follow approaches similar to this for generating their maps.

As another example, in *No Man's Sky* each of the planets in the universe are procedurally generated along with their associated biomes and lifeforms. *No Man's Sky* uses a "blueprint" system, where hundreds of basic templates for animals, plants, etc. are first defined by artists and designers. Then during the generation of a planet, the biomes and environment are first created using a set of rules. The planet and its biomes are then populated with instantiations of the base templates for animals and plants. The instantiations are also made using a set of rules to ensure consistency across creatures/biomes/plants/color palettes/etc. In this case the designers and artists encode their knowledge into various templates, content blocks, and systems of rules, but encoding rules and creating templates can be very difficult to get right. This can be seen with the improvement of *No Man's Sky* post-release, partially due to the designers tuning the rules and templates.