UNIVERSITY OF CALIFORNIA
SANTA CRUZ

RECOMPOSING PROCGEN

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

DOCTOR OF PHILOSOPHY

in

COMPUTATIONAL MEDIA

by

Isaac Karth

March 2023

Table of Contents

List of Figures

Abstract

Recomposing Procgen

by

Isaac Karth

Procgen is the field of making things that make things. Also known as procedural content generation (PCG), and overlapping with generative design and generative art, procgen is a growing field of inquiry in academic contexts and the focus of thriving scenes of craft practice. Procgen is frequently approached as if it was about getting something out of nothing. By contrast, this dissertation is founded on the observation that all information that comes out of a generative system must first have been inserted into the generative system, including information encoded into it by virtue of how the system was assembled from parts. Interpreting the compositional parts of a generative system, we can trace the relationship between inserted input and emergent output. Understanding how the modular compositional operations work in isolation makes it easier to recombine them into new generative systems.

My dissertation investigates the modularity of procgen via three research questions centered around aesthetics, composition, and applications. How does the structure of a generative system impact the aesthetic effect of the output? What are the lower level components that make up a generative system, and how do those components fit together? What new applications of generative systems are unlocked by taking a compositional view of their structure?

My work recomposes our collective understanding of procgen by building technical systems, applying humanistic interpretation to understand those systems in practice, and illuminating the connections between what we build and how we interpret what we build. I apply this method to existing generators, including some seen as canons in the field. I also assemble new generators using new capabilities emerging from the artificial intelligence literature. Taken as a whole, this dissertation establishes a practice of recomposing procgen: taking apart our understanding of procgen and putting it together again in new ways.

In memory of AnnaLeah and Mary

Acknowledgments

First of all, I want to thank my advisor Adam M. Smith for being with me on this journey, right from the moment we started a research project by collaborating on the beach. You deserve more credit than you have claimed.

I want to thank Kate Compton for her foundational work and subsequent discussions. This dissertation would not exist without her prior work. We need to write about liquid and solid generativity.

I want to thank Noah Wardrip-Fruin and Michael Mateas; without you I wouldn't have come to Santa Cruz in the first place.

I'd like to thank the Seabright Camerata for all of the discussions, debates, dialogues, and discourse. I wouldn't have been able to do nearly as much research during the pandemic without you. Good morning, floating shard.

I want to thank Max and Nic for being the best of all possible roommates (in all of the most trying circumstances). Nightly research conversations are essential for generating ideas, and just in general I don't think I'd have made it through the past five years without you.

I want to thank Jasmine Otto; research would have been much less exciting without you around to share ideas with. I'd like to thank Tammy Duplantis for all of her help; we should do more artistic collaborations. I'd like to thank Barrett, Batu, Saya, Ross, Ken, Zooey, and all of the other graduate students in the Design Reasoning Lab.

Thanks to Gillian Smith for being on my committee, general feedback, and her pioneering research in general, which cannot be measured in terms of linearity and leniency.

My thanks to Josh McCoy and Ben Samuel for the conversation in Alberta (and other times, but especially that). It meant a lot to me.

I want to thank Ian Horswill for his helpful comments. I also want to thank Mike Cook for our conversations and ProcJam in general. We should hang out more.

I want to thank Ian Schreiber for useful advice at a critical time, just before entering grad school again.

I want to thank my parents, Jerry and Marianne Karth, for their support through it all.

I want to thank all of the Computational Media grad students for all of our discussions, and everyone in the generative seminar.

I want to thank the Science Internship Program for the funding support, and all of the highschool interns for your eager participation in research. We won an award [189]!

I want to thank Amit Patel and Tarn Adams for our conversations about games and programming.

I want to thank Alex and Shawn. We should hang out again.

I want to thank my fellow Graduate Student Association officers during the year of fire: Yulia Gilichinskaya, Tony Boardman, Abigail Walsh, Veronica Hamilton, Sarah Mason, Rora, Jane Komori, Jack Davies, Rowan Powell, Natalie Ng, and Clara Qin. I equally want to thank the subsequent GSA officers I served with: Londen Johnson, Rora, Daniel Copulsky, Rora, Tomas Ocampo, Jazmin Benton, Rora, and Stephanie Herrera.

I want to thank, Brynna Downey for her work on SFAC and everything else.

I want to thank Peter Biehl, who still owes me a drink and I intend to take him up on it.

I want to thank Lori and Cindy for starting the wildcat strike.

This research was conducted during power outages, a flooded apartment, termites, a historic wildcat strike, a global pandemic, wildfire evacuations, a historic statewide union strike, and a couple of nondescript earthquakes.

0.1 Contribution Statement

The text of this dissertation is based on the following previously published material:

- Karth, Isaac. 2019. "Preliminary poetics of procedural generation in games." In: *Transactions of the Digital Games Research Association 4.3.* doi: `https://doi.org/10.26503/todigra.v4i3.106`. [183]

- Karth, Isaac. 2018. "Preliminary poetics of procedural generation in games." In: *Proceedings of the 2018 DiGRA International Conference: The Game is the Message.* url: `http://www.digra.org/wp-content/uploads/digital-library/DIGRA_2018_paper_166.pdf` [182]

- Kreminski, Max, Isaac Karth, and Noah Wardrip-Fruin. 2019. "Generators that

Read." In: *Proceedings of the 14th International Conference on the Foundations of Digital Games.* ACM. doi: `https://doi.org/10.1145/3337722.3341849`. [200]

- Karth, Isaac and Adam Marshall Smith. "WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning." In: *IEEE Transactions on Games.* doi: `https://doi.org/10.1109/TG.2021.3076368`. 2021. [188]

- Karth, Isaac and Adam M. Smith. 2019. "Addressing the Fundamental Tension of PCGML with Discriminative Learning." In: *Proceedings of the 14th International Conference on the Foundations of Digital Games.* FDG '19. San Luis Obispo, California: ACM, 89:1–89:9. isbn: 978-1-4503-7217-6. url: `http://doi.acm.org/10.1145/3337722.3341845`. [187]

- Isaac Karth, Batu Aytemiz, Ross Mawhorter, and Adam M. Smith. 2021. "Neurosymbolic Map Generation with VQ-VAE and WFC." In: *Proceedings of 16th International Conference on the Foundations of Digital Games.* doi: `http://doi.org/10.1145/3472538.3472584`.

As a general rule, the research was supervised by my co-author Adam M. Smith. Noah Wardrip-Fruin additionally supervised some of the research. My co-author (and roommate) Max Kreminski collaborated on the generativist reading research, as well as contributing to discussions on and around many other research projects. My co-authors Batu Aytemiz and Ross Mawhorter were deeply involved in the implementation and training of the neurosymbolic map generation system.

All images are my own, except where otherwise noted in the captions.

Chapter 1

Introduction

The central thesis of my dissertation is that all information that comes out of a generative system must first have been inserted into the generative system. Interpreting the compositional parts of a system, we can trace the relationship between inserted input and emergent output.

The title of this dissertation, *Recomposing Procgen*, refers to this compositional method of analysis. Procgen is an umbrella term in common use for things that make things,[1] in recognition of the intermingled and cross-disciplinary nature of Procedural Content Generation (PCG), generative design, procedural modeling, generative methods, and generative art. The more inclusive category of *procgen* makes the commonalities apparent.

The frequent temptation for researchers is to treat each generative system as a monolith, sorting it by its purpose or algorithm (e.g., Togelius et al. [354]). But when

[1]Borrowing the definition popularized by Mike Cook and ProcJam: “Make something that makes something” [72].

we look closer, generative systems across all categories are composed of modular parts that can be recomposed. Very different systems can share similar components, complicating the monolithic labels. Hence, “recombining,” emphasizing that these system are composed of modular parts that can be reconfigured—recombining them into new systems.

In light of the compositional nature of generative systems, this dissertation is an attempt to answer three research questions:

- RQ1 (aesthetics): How does the structure of a generative system impact the aesthetic effect of the output?
- RQ2 (composition): What are the lower level components that make up a generative system, and how do those components fit together?
- RQ3 (applications): What new applications of generative systems are unlocked by taking a compositional view of their structure?

Together, these research questions point towards a framework that we can use to understand how the compositional nature of a generative system encodes and transforms information.

In answering these questions, I’ve drawn on the research that I’ve done over the course of time time here at UCSC, the generative design classes I’ve helped teach, the generative systems I’ve made, and the many years I’ve spent writing about the vast array of generative projects that I’ve encountered.

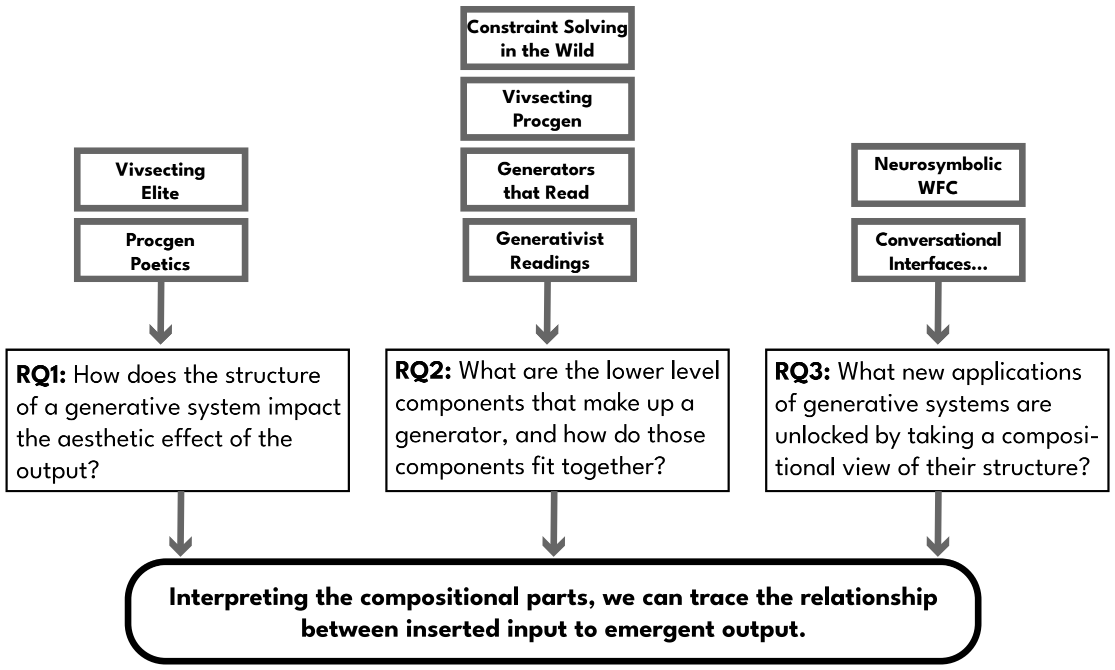

Constraint Solving in the Wild
Vivsecting Procgen
Generators that Read
Generativist Readings
Vivsecting Elite
Procgen Poetics
Neurosymbolic WFC
Conversational Interfaces...
RQ1: How does the structure of a generative system impact the aesthetic effect of the output?
RQ2: What are the lower level components that make up a generator, and how do those components fit together?
RQ3: What new applications of generative systems are unlocked by taking a compositional view of their structure?
Interpreting the compositional parts, we can trace the relationship between inserted input to emergent output.

Figure 1.1: The relationship between chapters and the three research questions

Working with students has been particularly illuminating, forcing me to confront the assumptions that people have when they are encountering these ideas for the first time. I've learned as much from my students as they've learned from me.

1.1 Composition

To understand how a generative system works—or how to construct one ourselves—we need to understand how information is inserted and transformed. By deconstructing generative systems into their components, we can start to ask questions about the specific ways that a given generative system is transforming that information.

There are many past attempts at describing how generative systems work, from monolithic models to more modular frameworks. In Chapter 3, I examine the history of academic frameworks for describing procgen. There have been many attempts to describe partial frameworks for particular procgen approaches (e.g., PCGML). The more modular frameworks provide a more robust explanation of a wider variety of generative systems.

However, there is another approach to examining how people think about building generative systems. After all, people have been building generative systems for a long time. And moreover, they have been building tools to help them make generative systems. Chapter 4 is dedicated to looking at how generative-system-making tools express ideas about what generative systems are and how they can be constructed. Examining the way different tools represent the compositional nature of generativity grounds our understanding of generativity in practice.

1.1.1 RQ1: Aesthetic Effect

I started my personal inquiry with the question of effect; specifically, what aesthetic effect do we experience when a generative system operates in one way rather than another. Chapter 5 addresses this, and points to the further question of what aesthetic a generative system can express in general. This question turns out to be more critical to our understanding of generativity than I first expected.

Generativity leads to significantly different aesthetic effects than non-generative artifacts exhibit. Even when a generative system produces an artifact that is identical to a hand-made object, the generated artifact has very different poetics.[2] Rather than

[2]If Pierre Menard re-creates the *Quixote* line-by-line, is it a different work? [35]

merely recreating hand-made objects, the poetics that are unique to procgen give us leverage to employ procgen to specific ends, creating effects that would be impossible to create by hand.

Equally, there are things that generativity can't do. Knowing when to ground a generator in a hand-made template or how to include outside data is a major part of designing any generator.

Ultimately, the poetics lens can tell us why one approach to generation gets better results for a specific application. *Elite* is an early example of the use of procgen in games, to the point that it frequently mentioned in the introduction of many research papers. Looking at it through the lens of compositional poetics, the lasting impact of the generator in *Elite* comes from the way it modulates the generator's focus over the different scales that the player progresses through in the course of play (Chap. 12).

1.1.2 RQ2: Connecting Components

How the components connect can be looked at from two points of view. Looking at specific components through the lens of their operational purpose lets us study the context they operate within. And looking at the topology of the network tells us about the general patterns.

The overall view of a generator can be understood through the lens of what I've termed vivisection (Chap. 8). Breaking the monolithic generator into component operations shows how the small effects of operators doing mere generation add up to create more complex generative effects. Classifying operations by how they transform information lets us map how information is inserted into the system, demystifying the process of constructing a generative system.

Turning to specific operations, one category has been generally overlooked in the generative context: reading operations that interpret complex inputs. There are many humanities-inspired lenses that can be applied, but part of the point of Chapter 6 is that reading is a deliberate process of translation, with no objectively right answer. Generative systems are inherently messy and hard to measure.

Continuing the theme of looking at procgen as it is practiced, Chapter 9 looks at a generative system in operation. By interpreting a technique from the non-academic game-development community through the lens of constraint solving, I introduced the game development community to the larger implications of the algorithms they were using and connected the academic community to applications of procgen research outside of the ivory tower. In addition to the constraint solving, WaveFunctionCollapse further introduces a reading process, learning the constraints from an image.

Breaking down *Elite* into its component operations allows for much deeper analysis than categorizing its algorithms in monolithic terms (Chap. 12).

1.1.3 RQ3: Applied Compositional Lens

Applying the lens of the compositional framework to an existing generative system reveals opportunities for intervention. This understanding of how the system works on a modular level can be used to pinpoint parts of the generative system for potential intervention—in this case, to enable us to speak to the generative system designer in the same language it uses to generate the individual generative systems (Chap. 10).

Continuing this line of research, we can further disassemble WaveFunctionCollapse and recompose it with a Vector-Quantized Variational Auto-Encoder to produce a neurosymbolic generative system (Chap. 11). By applying our topological understanding from the vivisection of both the VQ-VAE and WFC, we can combine the two methodologies into a unified system. This goes beyond the academic/non-academic divide and bridges the gap between different academic communities.

Reversing things, this compositional framework has another use. Instead of using it to analyze generative systems, we can instead use the techniques in building generative systems as a method of critique (Chap. 7).

Finally, *Elite* is frequently mentioned as an early example of procgen, but is sometimes framed as an anomaly because it was using procgen for compression. Modern games don't have to deal with such tight platform limitations, making compression less of a focus in the discussion. But the point of the compositional framework is that procgen is inherently about information, and viewed through this lens *Elite* is a prime example of the relationship between data compression and procgen in general (Chap. 12).

Chapter 2

Information in Procgen

We may hope to make all things new through the creativity that is left, but in a world of chance there will always be an element of chance. Thus, while we may seek after the eternal and the holy, the temporal and profane will forever remain chained to its form.

`AI Religion Bot @gods_txt` [130]

The boldest claim in this dissertation is that a generative system does not create information. By this I primarily mean that the complexity of a generative possibility space is directly correlated to the complexity of a generative system and its parameters.

Often procgen is presented in terms of autonomous creation. In the introduction to their textbook *Procedural Content Generation in Games*, Togelius, Shaker, and Nelson define procgen: “PCG refers to computer software that can create game content on its own, or together with one or many human players or designers” [355, p.1] and again, “The terms ‘procedural’ and ‘generation’ imply that we are dealing with computer procedures, or algorithms, that create something” [355, p.2].

The most foundational question that undergraduate students in generative design classes have asked me is, “How do I start making a generator?” It took me a while to figure out exactly what they meant by it, but I found it to be a particularly illuminating question. In essence, they were staring at a blank page and asking what to type to make the impossible magic happen. How could they create something out of nothing?

While I believe that when many practitioners use the word *creation*, it is frequently meant in a narrower sense, the general impression that we, as a field, have collectively projected is this idea of the autonomous machine that creates. The popular press speaks breathlessly of creating quintillions of planets via math [123], and my students are intimidated by the demands of divine creation.

2.1 The Content Arms Race

In 2005, Will Wright gave a talk at the Game Developer’s Conference [382], which is generally credited with reviving interest in procgen [355, p.3]. The ostensible title was “The Future of Content”, the big reveal was the in-development game *Spore*, and the general thesis was that the demand for content creation for games was outpacing developers’ ability to keep up. Data had became more important than algorithms.

The initial reaction is that procgen must be what lets us keep up with this “content arms race” [382]. Whether acting as a force multiplier for artists or creating content from scratch, the point of procgen is surely that it gives us extra content for little effort, creating something from nothing.

From this point of view *Elite* appears to be an early anomaly: the creators would probably have been happy making the content by hand, but they had to compress it because they could barely fit the game onto the disks in the first place (Chap. 12). Many academic discussions of the procgen in *Elite* mention that it used procgen to overcome platform limitations, and then go on to discuss other reasons to use procgen (e.g., [355, p.4]). This inadvertently gives the impression that the connection with compression is just a relic of the time. With the advent of the CD-ROM, surely the compression aspect of procgen is a thing of the past?

Except, in his talk, Wright focused heavily on the *demoscene*—the hobbyist generative artist scene that developed out of the early copy-protection cracking communities into virtuoso displays of maximum graphical effect achieved with minimal amount of bytes. The demoscene is a prime example of contemporary procgen that unabashedly embraces compression.

Wright was continuing the themes from his 2003 GDC talk, "Dynamics for Designers" [381], which focused on emergence. Wright's unofficial title for the talk was "Why I Hate Calculus...and why I Love Compression." And he summed up his thesis as "How we use emergence to engineer larger possibility spaces with simpler components" [381].

Is emergence the answer for where procgen gets its information?

2.2 Emergence

In games, emergence is often used in the sense of Jesper Juul's "structures of emergence", in which simple rules combine to create variation—in contrast with "structures of progression" that feature serially introduced challenges [174]. Juul notes that emergence was a popular topic at the time and cites a talk by Harvey Smith for defining it as "An overall term for situations or player behaviours that were not predicted by the game designers" [174].[1]

In "Engineering emergence" Joris Dormans questions Juul's perspective, seeking "strategies to merge structured level design and emergent, rule-based play more effectively" [90, p.14]. Dormans posits that the use of the term emergence prior to Juul is "often in reference to the use of the term within the sciences of complexity," where it is defined as, "behavior of a system that cannot be derived (directly) from its constituent parts" [90, p.16]. For procgen, this scientific sense is closer to why we care about emergence.

Dormans describes Stephen Wolfram's study of cellular automata[2] as an example of emergent behavior in a complex system [90, p.16]. This connection is widespread: cellular automata are frequently used as examples of how complex systems can emerge from simple rules. The most well-known cellular automata is probably Conway's "Game of Life" [121], though the concept predates Conway, ultimately tracing back to Stanisław

[1]Harvey Smith's exact definition: "You could define emergence as an event that occurs, but that could not have simply been inferred from a system's rules. Emergent behavior occurs when a system acts in an organized fashion beyond the sum capabilities of its individual parts" [329].

[2]For example, the 'Rule 30 Automaton' [378]

Ulam and John von Neumann [369, pp. 94–95]. Cellular automata have been widely used in generative art, including generative music, and are closely connected to emergence through complexity theory [46].

In complexity theory, emergence is a more complicated concept than the initial videogame research context. I present a brief summary here, drawing primarily from Joan Soler-Adillon's article "The Open, the Closed and the Emergent: Theorizing Emergence for Videogame Studies," in which he argues that emergence in game studies, particularly Juul's framing of the term, "falls short in capturing the potential of such an intricate topic" [334]. In particular, Soler-Adillon suggests that it allows us to distinguish between "self-organizing phenomena in digital games" and "the appearance of emergent novelty" [334]. To paraphrase, these self-organizing patterns result from agents interacting but produce new behaviors that cannot be understood through the analysis of the behaviors in isolation. In contrast, Soler-Adillon describes novelty as when "emergent pheonomena appear as new functions or behaviors in a known system" [334].

In Brian McLaughlin's depiction of British Emergentism, the original conception of emergence originally concerned causes that are not reducible to the laws governing lower levels of complexity (e.g., chemistry is not reducible to elementary particles, biology is not reducible to chemistry, etc.) [228, p.22]. The configuration of a structure exerts a fundamental force, unforeseen by the analysis of the individual interacting pairs of the less complex level; there must, therefore, be forces on the chemical level that are just as fundamental as gravity is on the physical level. While this was not unreasonable at the time, McLaughlin suggests that quantum mechanics led to the downfall of the original Emergentism: for the first time, chemistry could, in fact, be explained by particles, and particles by fundamental physical forces [228, p.23]. The reductionist point of view carried the day: each of the sciences could be reduced, in turn, to the more fundamental level, and ultimately to the universal physical laws. The whole can be explained as the sum of its parts.

Of course, the idea of emergence doesn't end there: Soler-Adillon traces the revival of the idea through Phil Anderson's critique of reductionism in "More is Different" [9] to the more contemporary debates on supervenience and downward causation [334]. Soler-Adillon suggests that the British cyberneticans laid the groundwork for the later return of emergence, as part of the complexity sciences, chaos theory, and particularly artificial life [334]. Additional levels of complexity don't exhibit new fundamental laws, but self-organizing behavior does give us results that are difficult to predict from the component parts. Soler-Adillon contrasts the authors who understood emergence as self-organization, and those, like Peter Cariani, who "developed his theory of emergence-relative-to-a-model in part as a critique of more ambiguous formulations within the field," and instead focusing on "emergence as a generator of novelty." [334].

Dorman and Soler-Adillon both reference Jochen Fromm's taxonomy of emergence. Fromm's taxonomy is "based on different feedback types and the overall structure of causality or cause-and-effect relationships" [111]. Fromm points out that since "the cause is normally the unclear point in emergence" this makes for a natural taxonomy [111].

Type I has only feed-forward relationships; Type II has simple positive or negative feedback; Type III has multiple feedbacks, learning, and adaptation; and Type IV is multi-level strong emergence, "which cannot be reduced, even in principle, to the direct effect of the properties and laws of the elementary componets" [111]. Fromm is particularly concerned with boundaries and jumps between levels: the essential taxonomic element is the feedback between levels.

Soler-Adillon disagrees with Fromm's classification on two fronts: first, that the more fundamental distinction is between self-organization and the generation of novelty; and second, Fromm's inclusion of intention as part of a feedback loop: Soler-Adillon points out that, despite the name, acting with intent is definitionally not self-organization [334]. He also notes that strong emergence—emergence which cannot, even in principle, be derived from the constituent parts—is a controversial topic.

Weak emergence, as described by Mark Bedau, is when the macrostate's microdynamic can only be derived by simulation.[3] This does not involve the introduction of a cyclical downward causation, unlike strong emergence.

Ultimately, this question does not concern us in the context of procgen: In either case, the emergence is dependent on and would not exist without the constituent parts, regardless of irreducibility.[4] And as a practical matter, only systems that can be

[3]He expressed it as: "Macrostate P of S with microdynamic D is weakly emergent iff P can be derived from D and S's external conditions but only by simulation." [23, p.378]

[4]Strong emergence, if it exists, *might* allow for the creation of information, but while that information would be irreducible to the less complex model, it would not be independent of it, and therefore the general point holds. Still, it would complicate the modular view, which is essentially, though not exclusively, reductionist. Reductionism takes the position of the information theory principle that we can have information compression but not the gain of new information. This has led to debates about how consciousness operates. The consciousness researchers Thomas Varley and Erik Hoel suggest a possible resolution is that the reason the sciences resist reduction is that emergence can be explained in

simulated can be used for procgen. Therefore, a generative system describes a possibility space that, while unpredictable, does not exceed the information that has been inserted into it. This is, of course, usually not obvious: the point is that the emergent complexity can be complex enough to defy easy pattern recognition. The magic trick is where the information comes from.

For Wolfram, cellular automata are a prime example of the power of emergence:

> But where did that complexity come from? We certainly did not put it into the system in any direct way when we set it up. For we just used a simple cellular automaton rule, and just started from a simple initial condition containing a single black cell. [378, p.28]

The initial conditions are certainly simple. The problem for our question of emergence is that these simple cellular automaton rules describe systems that are Turing complete [71]. Though it appears simple, a cellular automata is a complete program. This might seem surprising, but there is precedent: Turing's original description of a Turing machine, while slightly more complex, is still quite simple: a machine with states (m-configurations) and a tape with squares that are "each capable of bearing a 'symbol"', and the finite set of instructions [360].

It is certainly astonishing that Turing machines can be so simple and computers can produce such complex systems with emergent results. But for our specific question of emergence in procgen, inserting an entire computer into our apparatus is the furthest thing from creating information out of nothing.

terms of information *conversion*—information on one scale is a different type than that on another [363]. However, this is an ongoing debate. If a procgen generative system achieves consciousness it may be important to revisit it, but a conscious system is arguably outside of the realm of procgen: a non-deterministic generator with free will requires an entirely different set of research questions.

Inventing a formula you don't have is laborious. Calculating the Mandelbrot fractal is much easier when you have the formula to compute it and the process that computes it, which, when executed will unfurl into beautiful patterns. Cellular automata exhibit complex properties as emergent effects of simple rules, as you might expect from computation. But just because you discovered or built a computer doesn't mean that computation is information created out of nothing, in the sense that it is meant in this chapter. All the information about the fractal is expressed in the formula, and the emergent result cannot exceed that possibility space.

Whether it is a mathematical formula you discovered, or a computer you built out of rules, the emergent effects can be traced back to some process that you added to your generative system. Emergence doesn't create information from nothing. Both the processes and parameters of the generative system are information that has been added. We tend to focus on the parameters as input, because that is the obvious injection of information that the typical user experiences. However, the construction of the processes is also information, and often the more important source of the two. My term for this construction, where you encode something into a generative process that can create it, is a *generativist reading* (Chap. 7).

There are many generative operations that operate on either purely random numbers (such as a random seed) or simple continuous ranges of numbers—for example, Perlin noise, which uses the coordinate space as the input parameters. The heart of a generative operation is what the process does with that completely random or completely ordered signal to transform it into interesting generative complexity (Sec. 5.3).

This is why emergence is important for procgen: a simple process that emergently generates a complex result is exactly the kind of operation that we need to construct our generative systems. Remember, information must come from somewhere. Imagine our generative system as a machine, with a hopper on top for input and a chute at the bottom for output. We can insert information into a generative system from the top—as parameters to be fed into the machine—or from the side—installed as part of the machine. A process with an emergent effect can be a compact addition to our generative machine.

The more processes that we have in our toolbox that possess these useful properties, the easier it is to build effective and expressive generative systems. Emergence isn't a magic wand. Rather, it's a wrench-and-socket set, where emergence is what produces the leverage to make our tools work. Collecting emergent systems is useful for practical procgen, because each emergent process becomes another potential tool in the toolbox.[5]

When we take the modular view of a generative system, all of the emergent effects are dependent on the individual parts—though, of course, in non-obvious ways. A large part of the study of procgen, then, is to build the connections that let us understand the emergent effects in implementable and understandable ways, so that we can analyze their use in generative systems and effectively incorporate them into our

[5]One practical issue we run into when constructing a generative system is that humans are good at pattern recognition. Kreminski suggests that there is evidence that suggests that people navigating a generative possibility space as a creative act quickly learn to understand the artifacts produced by the generator in terms of the possibility space [199]. One way to counteract this is by using new configurations of generative operations that can create new patterns, extending the creative arms into a higher order space.

own generative systems.

2.3 Compression

The use of procgen for compression in *Elite* is sometimes viewed as an anomaly. However, we can view many contemporaneous generative systems through the same lens of compression and derive useful and practical applications of procgen.

Most generative systems incorporate the idea of a random seed: a number that the generation starts from. The seed is arbitrary, but allows a random generation to be repeated: run the same sequence with the same seed, and you get the same result. *Elite* is no different: everything it generates starts from the seed 5A4A 0248 B753 (Sec. 12.1.2). Transforming this seed via the generative system produces complete galaxies of stars, planets, and their economies.

Does a planet exist before it is generated? From the compression perspective, the random seed that the planet generator uses is merely the compressed version of the planet. With a deterministic generator, you'll get the exact same planet if you extrapolate it from the seed or store every last detail about it on disk.

The difference, of course, is that the more verbose format can store a larger variety of planets, and it is easier to have the result of complex and interconnected systems that way. In practice, it is mostly a matter of how complex the generator is; a sufficiently complex generator can reproduce a specific desired result—in the extreme, by just including it as a template—and therefore the problem is the expressive range of the generator, not that it was generated per se. There are many planets that could be expressed in the format of the final system that do not exist: if Elite generates a planet with a name that has an odd number of letters, it must include the letter 'A' (Sec. 12.1.5).

Perlin noise can be viewed similarly: it computes a value for a given x, y point, converting a continuous space of 2d (or nd) coordinates into a more complex space. The input is simple, but the reason to use it is that it creates the perception of natural and unpredictable noise that, nevertheless, exhibits the property that points which are near each other have similar values. This imitates many natural phenomena closely enough in its end result to be useful for generating the ontogenic form of many things (c.f. Sec. 6.2.2.1).

In either case, the generative process takes parameters that are either excessively disordered and random or excessively ordered and fixed and translates them into a form with more effective complexity. This generative complexity is at the heart of how procgen produces aesthetically compelling results (Sec. 5.3).

2.4 Conclusion

Generative systems do not create information. A great deal of the rhetoric around procgen certainly implies that the purpose of using generativity is to get something out of nothing.[6] And generating an artifact can certainly seem like a magic trick.

[6]One way this is implied is by the frequent speculation about cost-savings associated with building a generator, e.g., "Algorithms that can produce desirable content on their own can potentially save significant expense." [356]. There has been very little research into quantifying the actual cost-savings,

But taking the rabbit out of the hat requires that we put the rabbit in the hat in the first place.

The obvious way that information enters a generative system is through the parameters we give it. In the vast majority of generative systems, this input appears to have less information than the rich output of the generative artifacts, making it appear as if we got something from nothing. If we think of it as some kind of steampunk contraption of gears in a box, it looks like we feed our random seeds[7] into the hopper as individual numbers and take out a fully formed rabbit from the hatch at the bottom. However, when we open the hood and look at the individual components, we get a different picture.

Most components in a generative system include a lot of information about the kind of artifact that they help produce.The ostensible information flow of a Perlin noise [267] generative operation is to convert coordinates in space into a point that is similar to, but different from, neighboring points. However, a Perlin noise operation inherently includes the information needed to perform that calculation. The gears that make up the machine are also information, and are just as significant as the parameters we set.

Going further, the arrangement of the gears is, itself, a way to represent in-

and a fair amount of anecdotal evidence to the contrary: “thanks to procedural generation, I can produce twice the content in double the time” [351]. Procgen requires that you pay the cost of encoding, and the cost savings can only come if that is less than the cost of building everything by hand. Personally, whatever the actual cost savings, I’m far more interested in using it to build things that would be impossible without procgen. There are effects that procgen produces we can’t get any other way. For more on these effects and other alternative motivations for using procgen, see Sec. 5.8

[7]A *random seed* is the initial number that gets transformed by a pseudo-random number generator (PRNG) into an unpredictable new number, and forms the basis for much of the randomness in procgen.

formation. The effect of an assemblage of operations working alongside each other can be greater than the sum of the individual operations. They are, after all, programs, whether they are being run on a digital computer or not.

Emergence can roughly be defined as the effect being more than the sum of the parts. We frequently leverage it as part of constructing a generative system. However, emergence is not free information. Even though it frequently isn't obvious how the emergent properties of a system relate to the underlying components, they are still dependent on those components. Emergence, therefore, is a red herring when it comes to the question of information with respect to procgen: emergence is an effective way to decompress a simple representation into a much more complex system, but it doesn't create brand-new information.

While it takes particular skill to have the correlations in the machine match the *specific* outcomes we want, the general phenomenon of emergent systems is so easy that most interesting systems do it accidentally. It can be difficult to stop a complex system from exhibiting emergence. One framework for understanding computer security is to treat the actual system, hidden behind the ostensible operations, as a *weird machine*, which can be characterized as the behavior of the actual finite state machine, rather than the intended finite state machine [93]. The emergent machine has properties that are undreamed of by the designer of the intended machine. A common challenge for procgen practitioners is, therefore, to seek emergent systems that have the desired correlations.

We can think of the way that a generative system uses information as a form of data compression. Our seed number doesn't magically turn into a rabbit. Rather, it flows through a set of rules for expanding that number into a particular rabbit. The magic isn't in the seed number, but rather in the particular workings of the machine that encodes the description of all possible rabbits.

We often take advantage of pre-compressed information. A common example is mathematical formulas. The infinite fractal detail of the Mandelbrot fractal [219] or the complex systems of a cellular automata appear to have complex patterns emerge from nothing. However, the information to create those patterns is inherent in the process. The program is as much a part of the procgen as the parameters are. Emergence is important because it is an encapsulation. Leveraging these mathematical tools is often an important part of designing a generative system.

Some of the other concerns my students had were about making something that was *generative enough*. Mere generation of simple things felt inadequate, and using a template felt like cheating. Viewed from an information perspective, this struggle comes from asking the wrong question. The appearance of magic comes as the system grows more complex but still has recognizable patterns in its output. There's no single component that will cause a system to be perceived as meaningfully generative.

Instead, the question becomes one of building information into the system's operation. Grounding a generator in specifics is a powerful way to inject information into the system. Simple things start mattering when they're combined in ways that express new ideas: we care about emergence in procgen because emergence is one of our tools to express ideas through the machine. The generative complexity comes from using the process to create associations. Perlin noise is meaningless until it is used to determine how the rabbit's fur bristles. Effective procgen is, therefore, often about expressing the relationships between concepts.

Chapter 3

Interpretive Frameworks

Before I present a new model of how procgen systems operate, we need to examine the existing frameworks. How have people described generators in the past? What models, or partial models, already exist?

While there is no universally agreed upon formal model of procgen, there have been a number of attempts to construct frameworks. Though this chapter covers the major frameworks and some of the earlier work that inspired them, it does not claim to be complete: in particular the generative art and computational creativity communities have been having parallel discussions.

With regards to the existing frameworks for describing PCG, there is a general conflation of three distinct lenses: the intrinsic properties of a given generative method, the way the generative methods being described are used, and the way that the architecture of a generative system is structured. This has led to, among other things, the common confusion between how a generator works and how it is used (cf. Sec. 8.6 and

Sec. 6.2.2.1)

One common way that this conflation plays out is in eliding the difference between the internal and external contexts of a generator. Many frameworks only consider generators in isolation, either overly identifying a particular generative system design with a single domain or, in the other direction, overlooking the context the generative artifact will be placed in.

Simultaneously, there is a general tendency to describe a generator as a monolithic entity, overlooking the interactions between its parts and therefore missing the opportunity to discover new ways of combining those parts or introducing new parts.

Isolating generators makes it harder for us to think about how our generators will be used or what context they will need to react to. Many projects incorporate external context, such as Cook's ANGELINA [74], so a general theory of procgen needs to account for how generators exist in context. The monolithic perspective makes it harder to reason about the ways that generators can be modified and obscures where implicit concepts can be represented as explicit data. This has been known to cause conceptual problems severe enough to cause major game development projects to stall for years [223].

The research that does rethink our assumptions about internal generator relationships has lead to new breakthroughs, discovering how inverting cause and effect can lead to better histories [136, 138], or using knowledge webs to track the player's awareness [167]. Notably, the insights mentioned came from developers in the wild rather than academic theoreticians, pointing to a critical failure in our academic theories. While research is larger than direct applications, we should be in a better position to turn our insights about how we describe generators into actionable designs than we currently are.

Parallel to this, game generation research has run into the problem of *orchestration* [207]: games are made up of many different interdependent parts that must be kept in alignment with each other to make a successful game. Frameworks that analyze generators as monolithic algorithms naturally have no way to approach this problem, and the orchestration problem is inherently about context as generators for different aspects of the game balance their outputs against other generators.

3.1 Vocabulary

The procedural generation field has numerous competing terms for similar concepts, to the point that the name of the object of study is contentious—procedural content generation, generativity, generative methods [66], procgen—partially due to attempts to stake out the boundaries of the field and the computational creativity attempts to escape charges of "mere generation" [365]. Across different fields the problem is even more acute [59, p. 91-92].

Before the widespread adoption of PCG as the term of art, there were many attempts to label the field: "Creative Generative Modeling" (1983), "Procedural Modeling", and "Procedural Generation of Geometrical Objects" [306] (1994), "Texturing and Modeling: A Procedural Approach" [95] (1994), "procedural generation" [96] (2000), "Procedural World Generation" [135] (2003), "Procedural Content Creation" [293] (2004), to note a few. Procedural Content Generation won out, but there are reasons to believe that it is not the best term for the field.

There has been some prior debate on the term; in particular Compton, Osborn, and Mateas argued that PCG is limiting precisely because of its over-emphasis on "content" and propose "Generative Methods" as a better name [66]. They trace the origin of the term to an Intel technical report in 2000, and describe the confusion of drawing the line between content and not-content.

For the sake of clarity, this dissertation standardizes on a set of terms informed by previous efforts to establish a common vocabulary, including "Search-Based Procedural Content Generation" [356]; "Understanding Procedural Content Generation" [323]; "Generative Methods" [66]; "Design metaphors for procedural content generation in games" [192]; and "A Generative Framework of Generativity" [64].

Therefore, the term I use for the field itself is *procgen.* I use the term *generative system* to name the things-that-make-things—the primary object of study. The things a generative system makes are called *artifacts* or *generated artifacts.* A *generative operation* is an encapsulated process inside a generative system, and modular units of selectable processes are *generative methods.* The set of all possible artifacts a generative system might be able to produce (given different configurations) is the *expressive range* [325], while a *generative space* is the possible output of a *specific* configuration of the generative system and its input parameters [325].

The person who constructs the generative system is a *generative designer*,[1] the person who interacts with the generative system is a *generative participant*, and the person who uses the artifact produced by the system is an *artifact user*, or *player*. Of course, all three roles can potentially be the same person.

3.2 Search-Based Procedural Content Generation: A Taxonomy and Survey

One approach that has been used to define a model of procgen is to describe a taxonomy. One of the most influential examples of this can be found in “Search-Based Procedural Content Generation” by Togelius et al. The primary focus of the paper drills down on a specific generative method: search-based evaluation and filtering [356]. By placing an evaluation function at the end of a generative pipeline, the generated artifact can be assigned a fitness measure. With a lack of prior groundwork to build on, the paper is forced to construct its own framework to then extend. Performing this foundational work has earned it its position as one of the most-cited surveys in the Procedural Content Generation field. The taxonomy presented is repeated (with additions) in the first textbook in the field, cementing its influence.

Its original context is as a “basic taxonomy of approaches” [354] concerned with establishing a context for search-based generation. Taken far beyond this, its limitations have affected the way we think and talk about the field.

[1]See also the discussion of roles in “Design Metaphors” [192]).

The paper itself acknowledges this limitation:

> "Note that these distinctions are drawn with the main purpose of clarifying the role of search-based PCG; of course other distinctions will be drawn in the future as the field matures, and perhaps the current distinctions will need to be redrawn." [354]

However, due to the shortage of taxonomies and surveys in the PCG field, it has become a frequently referenced source[2] and subsequent taxonomies have continued to be based on it. In 2016, Togelius, Shaker, and Nelson presented a revised taxonomy in *Procedural Content Generation in Games* [355].

We are overdue for redrawing the map, but we should take a moment to consider the taxonomy as it was originally presented: a set of paired distinctions, each pair forming a continuum. The distinctions drawn are online vs. offline, necessary vs. optional, random seeds vs. parameter vectors, stochastic vs. deterministic generation, and constructive vs. generate-and-test. In the later textbook formulation, the additions are "degree and dimensions of control", "generic vs. adaptive", and "automatic generation vs. mixed authorship" [355]. I'll examine these in more detail in a moment.

The unavoidable weakness of this taxonomy as a framework is that it is trying to fulfill two disjoint goals. The search-based algorithms part of the paper focuses on the structure and intrinsic properties of the search-based generative system: in other words, how it *works*. The taxonomic distinctions, in contrast, are more about how the generator is *used*.

[2]For example, as of 2022 Google Scholar lists it as having been cited nearly 800 times: `https://scholar.google.com/scholar?cluster=11372022422216892337` Archival link: `https://web.archive.org/web/20220929004121/https://scholar.google.com/scholar?cluster=11372022422216892337`

It is worth noting that, when it comes to the discussion of search-based algorithms specifically, the paper presents them as a subset of generate-and-test algorithms. Search-based generation adds a grade for the generated artifact via a fitness function and uses the context of past fitness scores when generating future artifacts. The paper makes a further distinction between the content representation and the search space, the genotypes of the internal representation of the generative model and phenotypes of the artifact, and point out that the phenotype of the artifact is the thing being evaluated by the fitness function. The revised version of the search-based approach presented in *Procedural Content Generation in Games* (PCGiG) has more breathing room and gives search-based generators their own chapters. This allows for a more in-depth examination of evolutionary search algorithms and content representation [354].

3.2.1 Search-Based Taxonomy Reexamined

Offline versus Online is the first distinction. The point at which the generation takes place is an important consideration, but it is also highly situational: the exact same generative system can be used both online *and* offline. Or, as is also common, different *parts* of the generator can be online and offline. One of the well-known early examples of procgen in games, *Elite*, includes both generation that happens while text characters are in the middle of being written to the screen *and* a substantial offline generative process that significantly effected the character of the game (Sec. 12.1.2).

The distinction between online and offline can matter for a discussion of gameplay: for example, if we use the player's current level to determine the difficulty of the next dungeon level we generate (as several classic roguelikes do) then it makes a difference when the generator was run. But the architecture of the generator can stay the same: this is a question of use but not of coding decisions. Other reasons to use offline generation include using it as part of a development tool (such as SpeedTree: cf. Sec. 5.6.2) or to manually moderate the content before release.

Where the architecture of the software does matter is when performance is a consideration: some generators are too costly to run in real-time. However, what counts as "real time" shifts as computational performance improves and architectures shift: an approach that uses massively parallel computation of matrix transforms has gone from unthinkably expensive[3] to a few milliseconds.

This gets even more interesting when we consider mixed-initiative generation. The user in a mixed-initiative system can be considered to be just another evaluation node (cf. Sec. 8.4). We can extend this idea to encompass online generators: the player is a processing node in the generator. This doesn't necessarily apply to all online generators: our roguelike dungeon level example from earlier is merely triggered by the user and receives no further input. But for the online generators that do incorporate player or generative user feedback, the classification that is more practical is to consider the player as a node in the system, rather than thinking about the system as a whole as being online or offline.

This also leads us to think about the situations where part of the system is

[3]A 2006 Game Developer article can speak in awe of how impractical a calculation of shattered glass is that requires a "12 teraflop, $110 million dollar supercomputer" [374] while in 2022 the state-of-the-art $1100 desktop GPU has three to four times that power and uses it to render reflections in Minecraft.

online and part is offline, as well as confront the idea that the latent artifact space of a generator can be considered to be simultaneously offline (because it exists mathematically, as defined by the generative system's ironclad rules) and online (because the particular artifact hasn't been extracted from the latent space yet). Many games exploit this as part of their generation, for example by saving a set of known seeds that describe particular parts of the latent space.

Therefore, online versus offline is primarily a consideration of *how to use* a particular generative system that we're building. It is certainly not an intrinsic property of a given algorithm or generative method. Blurring the lines further, the same generator can be run offline to identify interesting or acceptable seeds, and then run online using the discovered seeds as the starting point (which is the method used by *Elite*, Sec. 6.2.2).

Necessary vs. Optional generation is ultimately a distinction about the *artifacts* the generative system produces rather than anything intrinsic to the generator itself. It is important for how the generator is used and the player's aesthetic experience of the generator, but it is only indirectly a consideration in the architecture of the generative system itself: a design goal, but not a topological feature.

Part of the reason for the emphasis in the search-based taxonomy is that Togelius et al. discuss generators that *fail*, which is an important consideration for search-based generators but is not universal. Considering it as distinction on its own, all necessary content is not-allowed-to-fail, but some not-allowed-to-fail content is not necessary. It is also related to the distinction between generators that are correct-by-construction and those that are correct-by-testing (contrast with Sec. 8.1.3).

It has been pointed out that this can be seen as a distinction between functional (which affects gameplay) and cosmetic (which does not) game content [344, p.258]. Using this gameplay/non-gameplay as a distinction has long been a temptation in the procgen field. However, this is fraught, because in practice the cosmetic elements frequently have significant effects on the player (blocking sightlines, changing gamefeel, etc.)[4] More importantly, this tends to run up against the *aboutness labeling problem* (Sec. 8.6): a given generative system isn't intrinsically *about* anything.

Random Seed vs Parameter Vectors can be generalized: a random seed is just a specialized parameter, albeit one that often requires careful handling to ensure determinism. Further, the input to a generative system as a whole or to a generative operation by itself can include things outside this spectrum: e.g., while you *can* represent an image or a sound as an array of vectors, the meta-data that this is an *image* has its own meaning and implies things about what operations might be useful.

The taxonomy in *Procedural Content Generation in Games* renames this to **Degree and dimensions of control** and makes it about the amount of control that a practitioner has over the generator, from the very indirect control of specifying the random seed to supplying many parameters [355]. But we can add that search-based generation and constraint solvers offer an even greater level of teleological control (cf. 6.2.2), with the practitioner setting the desired result as a goal rather than manually fiddling with the parameters.

Stochastic vs. Deterministic Generation is framed as a question of gener-

[4]The *sound* of a weapon can have an effect: `https://www.youtube.com/watch?v=RDxiuHdR_T4` [117]

ator construction. However, it may be more useful to think of this question as a matter of tracing a *path* of determinism.

There's some confusion about random seeds in the original formulation: Togelius et al. exclude random seeds from the stochastic vs. deterministic spectrum, arguing not unreasonably that, "Note that the random number generator seed is not considered a parameter here; that would imply that all algorithms are deterministic" [354]. This is somewhat undermined by one of their examples of a game that has a completely deterministic generator being Elite, [38] which *does* use a random seed; it's just that it always starts with the *same* random seed, which was generated offline through a search-based generative approach [25, 37][5].

The revised taxonomy wisely simplifies this to just focus on the distinction without the distraction, but it is worth considering the paradox: all algorithms generally *are* deterministic except when influenced by an outside factor. But there are a lot of potential outside factors (using the system time as the seed, race conditions affecting the order of processing, and cosmic rays flipping bits during a speedrun being just a few examples). A deterministic generative system needs to be carefully engineered so that it doesn't introduce extra randomness; in contrast it is relatively easy to make a deterministic system unrepeatable by introducing randomness once.

This is, therefore, a useful engineering question: did we accidentally break determinism?

Constructive vs. Generate-and-Test is the most system-focused of these

[5]Specifically, Galaxy 1 uses a hexadecimal seed of: 5A 4A 02 48 B7 53 [25, lines 129–131]. See also (Sec. 12.1.2).

distinctions. A constructive system generates the content once, while a generate-and-test system generates an artifact, examines it, and if it doesn't meet the filtering criteria throws it away and generates another. Many kinds of generators elaborate on the basic generate-and-test model, including search-based generators. Purely constructive generators encode their rules in such a way to preclude invalid results: a Markov chain generator, to borrow Togelius's example, is only allowed to select from a set of valid possibilities. A distinction between generators that are correct-by-construction and those that are correct-by-testing is useful, because there are valid reasons to use both and knowing which one you are building can have a significant influence on design choices.

We can generalize this distinction: when considering other approaches, such as constraint solving (Sec. 3.4.1) and PCGML (Sec. 3.4.2), it becomes apparent that the main difference is in the *topological shape* of the generative system (Sec. 8.1.3).

The *Procedural Content Generation in Games* version of the taxonomy adds two more distinctions:

Generic versus adaptive is about the system adapting to its context. The discussion in *Procedural Content Generation in Games* is restricted to adapting to the *player*, mostly in the context of dynamic difficulty adjustment [164], emotional intensity (as in *Left 4 Dead* [31,352]), or player preference (as in *Galactic Arms Race* [151]). However, expanding from this narrow focus on players and games lets us consider a generative system's context more broadly: for example, generative audio artwork that responds to the current state of a building [349] or using ice-core data as part of a live performance incorporating spoken poetry and generative visuals [379]. Many generative systems include evaluation of complex data as a generative operation (Sec. 8.1.2, Chap. 6).

There's further useful discussion of this distinction in Smith's "Understanding procedural content generation" (Sec. 3.5) that discusses nuances of the adaptation and player interaction relationship.

Automatic generation versus mixed authorship is, as a taxonomic distinction, relatively straightforward: is there a human in the loop? While the popular image is that there is a pure form of procedural content generation that is fully automated, incorporating human decision-making or execution has been a component of many systems.

While the search-based taxonomy rightly points out mixed-initiative generative systems such as *Tanagra* [327], SketchaWorld [317], and *Ropossum* [307], it might also be useful to trace the idea of human-machine collaboration back Sutherland's SKETCHPAD, which prefigures this line of inquiry [346]. We can go so far as to claim that Sketchpad is a generative system *authoring tool* with heavy use of mixed authorship, similar to present-day generative system authoring tools (Sec. 4.3).

However, for our purposes a better way to model including a human in the generative system is as topological pattern rather than an absolute distinction (Sec. 8.4).

3.2.2 Generalizing the Search-Based Taxonomy

Togelius et al. consider some search-based-specific aspects of generation. However, some of these aspects of generative systems also apply more widely.

The authors draw a distinction in the mapping between the genotype and phenotype, between the “relative computational simplicity” of *direct encodings* and *indirect encodings* in which “the genotype maps nonlinearly to the phenotype” [356]. The terminology for how *genotypes* map to *phenotypes* is borrowed from evolutionary computation [356] but content representation is an important part of search-based PCG. Separate from the search-based taxonomy, I would argue that it is also applicable to designing PCG systems in a broader sense: even with a generator that is correct-by-construction, the designer needs to consider how the data is structured and how the artifact reflects that.

Both direct and non-direct encodings occur outside of search-based PCG. For example, a graph-grammar representation of a dungeon that is translated into tiles on a grid [91] is similar to Togelius et al.’s intermediate roguelike maze examples. On the other hand, the discussion of “avoiding the ‘curse of dimensionality’ ” [356] is more specific to search-based generators, but will be encountered as meta-problem when evaluating the expressive range of a generator in the process of its development.

This isn’t surprising if we consider the structural similarities between a search-based generator and a mixed-initiative generator. In the case of a simple mixed-initiative generator, the user is serving as both the evaluator of the artifacts and the variation of the artifact-construction parameters.

In the original paper, Togelius et al. divide evaluation functions into three classes: direct, simulation-based, and interactive [356]. While they only consider them in the context of search-based PCG, a design-time mixed-initiative generator can similarly be classified as an interactive evaluation function (see also Sec. 8.4). Direct evaluation functions score the observed features of generated artifacts and simulation evaluation functions that use techniques like having an agent interact with the generated artifact [356]. These are similar to the distinction between static and dynamic analysis in more general programming and computer science contexts.

3.2.2.1 Limitations of the Search-Based Taxonomy

Overall, the signal weakness of the taxonomy as a model is that it elides the difference between the generator-as-a-whole and the individual generative operations. This makes sense in its original search-based context, where the architecture of the generator could be assumed, but runs into difficulties extending it to a wider selection of generators.

The search-based taxonomy is a useful early model. Its limitations make it difficult to use as a starting point for building most new generators or analyzing generators that were built with assumptions significantly different from the original search-based context.

3.3 Search-Based Generation

"Search-Based Procedural Content Generation" has an important contribution, quite apart from the taxonomy: the discussion of search-based generation. And here we discover something interesting: Search isn't a *generator type*, it is a *topological pattern* in the system. Any generator can incorporate search as a part of its system architecture.

The basic search-based generator, as presented by Togelius et al., consists of a generate-and-test architecture with an evaluation function that evaluates an artifact with a grade (of one or more real numbers) rather than a binary result, and a content generator that is “contingent on the fitness value of previously evaluated content instances” [356]. Thus the next generation of artifacts is dependent on the evaluation of the previous generation of artifacts.

Viewed in a modular framework, search-based procedural content generation fits in neatly as a filter function: given the generative space of a generative system as input, search the space to find the system input parameters that maximize the fitness of the subspace that is filtered out of the larger generative space. The distinctiveness is not that a particular generative system is a search-based generator. Instead, the distinction is that the system uses a filter method in combination with a pattern of looping the evaluation back into the input parameters.

This is spelled out in the original paper, in which the difference between search-based generation, simple generate-and-test, and constructive generation are diagrammed as being defined by how the pipeline architecture is shaped (Fig. 3.1).

Here we see the true legacy of search-based generation: many other approaches to creating generators were inspired by it, directly or indirectly. While they have generally been regarded as separate and competing types of generators, in my view it is far more useful to view them as members of the same class: not generators but instead *filtering operations* (Sec. 8.1.3).

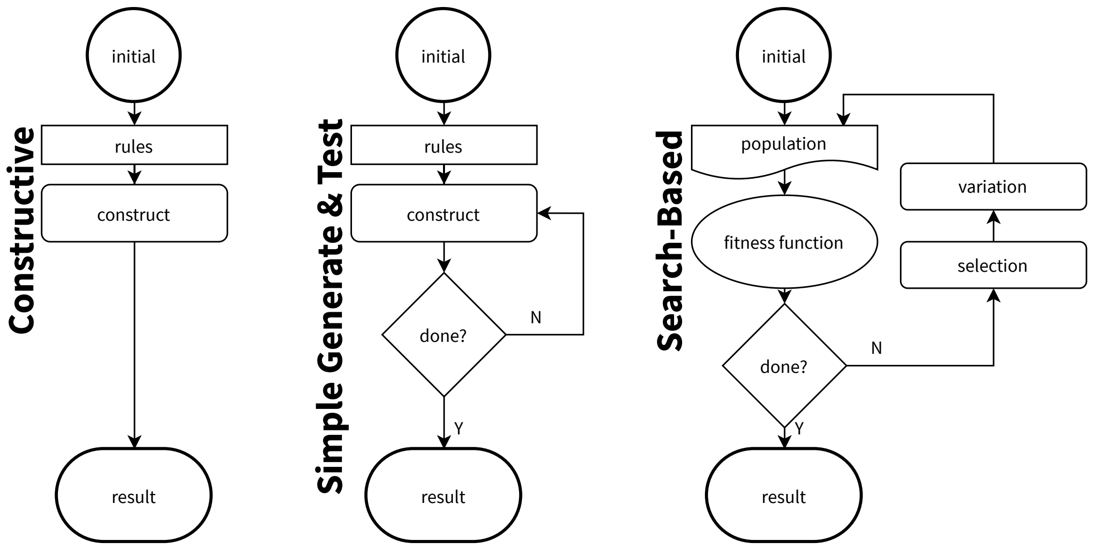


Figure 3.1: Comparison of different approaches to procedural content generation, after Fig. 1 in [356]: constructive, simple generate-and-test and search-based. Note that the topology of both generate-and-test and search-based approaches depends on introducing a loop into the directed graph so that the generative system can use the evaluation as a sensor for cybernetic feedback.

3.4 Other Topological Models Inspired By Search

By this point, “PCG via X” is a genre of research publications, a list which includes Search-Based PCG [356], Reinforcement-Learning PCG (PCGRL) [193], Machine-Learning PCG (PCGML) [344], and Deep Learning for Procedural Content Generation [209]. The weakness of framing these paradigms as separate areas of inquiry is that it obscures the architectural similarities between different paradigms and makes it harder for us to discuss the generative operations that make up these generators. This also tends to lend itself to describing generators as monoliths, unfortunately contributing to the popular rhetoric around the idea of a monolithic generator that will create everything[6]

3.4.1 Answer Set Programming for Procedural Content Generation

In "Answer Set Programming for Procedural Content Generation: A Design Space Approach," Smith and Mateas describe another top-level algorithm for PCG. In this case, applying a constraint solver to the latent design space of a generative system. The constraint solver acts as a filter (Sec. 8.1.3). By redefining the problem as one of generating answer sets, in this case through the "the declarative specification methods afforded by answer set programming (ASP)" [321] generative designers can leverage existing constraint solvers to "solve the meta-level problem of generator design" [321].

That meta-level turn leads to an important contribution of this paper: shifting the focus from individual artifacts to the entire latent space. By considering the latent space of both each operation and of the configuration of the generative system itself we can start applying generative techniques to the design of the generative system itself. By paying attention to the data flow rather than individual generative artifacts, we have additional insight into the anatomical features of the generator (Chap. 8).

It also touches on the difference in *time*, going beyond online/offline generation to consider how generate-and-test procedures can be refactored during *development* while preserving the design space. When combined with declarative programming, the timeline of the generative system's operation is unpredictable.[7] We can expand this to

[6]For more discussion of this, see also Kreminski's observation of how the procedural generation in Animal Crossing is discounted because it doesn't present as a monolith [201].

[7]c.f. the discussion in "Strange Loops in CFML" [319]

consider the generative system from multiple different time perspectives—the player's timeline, the meta-system developer's timeline, the generative designer's timeline, the generative system user's timeline, the dataflow execution order, the perspective of each latent space: a myriad of different perspectives, difficult to place on a single spectrum. This is more than a terminological sleight-of-hand. Complex generative systems naturally tend to operate on multiple timelines, as exemplified by the need to combine multiple intersecting types of content in game generation [207].

As a consequence of these perspective shifts, the cycles that the paper considers includes the exploration of the design space by the designer:

> In our method, the intent to generate artifacts from a (conceptual) design space is carried out by first modeling the design space with a logic program, and then invoking a domain-independent solver to produce answer sets which can be interpreted as descriptions of the desired artifacts. Experience with generated artifacts inspires redefinition of the design space. [321]

3.4.2 PCGML

"Procedural Content Generation via Machine Learning (PCGML)" by Summerville et al. focuses on the subset of procgen that uses machine learning. In contrast to other procgen approaches (constructive, search-based, solvers) which use hand-crafted parameters, PCGML instead uses machine-learned models to directly generate artifacts [344]. The paper argues that, while search-based systems might:

> "[...] use machine-learned models (e.g., trained neural networks) for content *evaluation*, the content generation happens through a search in *content space*; in PCGML, the content is generated directly from the model. By this we mean that the output of a machine-learned model (given inputs that are either drawn from a random distribution or that represent partial or previous

game content) is itself interpreted as content, which is not the case in search-based PCG" (italics in the original) [344, p.257]

While the authors acknowledge that corner cases exist, I would argue that, as with the other models, this unproductive discussion is a symptom of the monolithic framing. We gain very little by labeling an entire generative system as one or the other. On the other hand, we gain quite a lot by looking at the individual parts: when we acknowledge that a search-based generative system that incorporates machine learning as an evaluator (Sec. 8.1.3) is using both paradigms simultaneously, we can start asking questions about the commonalities between that algorithm and other machine learning algorithms.

The PCGML paper focuses on content that affects gameplay, pointing out that it has a stricter constraint of playability. Because PCGML can be trained on existing content, they point out that makes it especially suited for autonomous generation; cocreative and mixed-initiative design; repair of partial or broken content; recognition, critique, and analysis; and data compression [344]. PCGML techniques are categories by their form of data representation and training method. Data structure can be sequences, grids, and graphs; training methods are matrix factorization, expectation maximization, frequency counting, evolution, and backpropagation.

Despite the monolithic framing, the PCGML model is broadly useful: because it has such a well-defined scope (machine learning procgen) it can also be usefully applied to generative systems that use machine learning as one part of a larger whole.

One weakness is that some of the categories in the paper are extremely broad.

Specifically, backpropagation, which encompasses most of the recent machine learning tidalwave. Fortunately, a framework that addresses this also exists.

3.4.3 PCGRL

Khalifa et al. identified that the tension between level generation as a problem domain and reinforcement learning came from reinforcement learning being framed around agents making a choice in action space, while search-based generation is more usually framed as an optimization or supervised learning problem [193]. The solution was to reframe the generation process as "observation spaces, action spaces and reward schemes" that worked with existing RL algorithms [193].

Khalifa et al. claim that reinforcement learning has advantages over other methods because while machine learning can be faster than search-based methods because "no search is needed on demand" at the cost of the training time, reinforcement learning doesn't need training data and the incremental nature makes it easier to use in a mixed-initiative context [193].

They make a distinction between the space of game content (or in my terms, the latent space of generated artifacts) and "the space of policies that generate game content" [193]. Recasting the problem like this opens the possibility for other ways to think about how the generator itself operates.

The dual viewpoints of a generative system as simultaneously a series of choices in action space (cf. 12.4.1) and as a chain of operations lets us design systems by choosing which viewpoint is most useful for the problem at hand. Further, we can start to think about generative systems that themselves design generative systems. This becomes even more relevant when we encounter the discussion of deep learning below (Sec. 3.4.4)—the point of training the hidden layers in a neural network is, after all, to find the function that best matches the latent space we want to optimize for.

3.4.4 Deep Learning for Procedural Content Generation

While DLPCG and PCGML have some obvious overlaps, the authors of the deep learning paper frame their topic as having a narrower focus on specifically *deep learning* while simultaneously claiming a wider *application focus* on more types of content and more applications.

The five areas of content they identify are game levels, text, character models, textures, and music and sound [209]. The challenge they identify for deep learning is that game content needs to be functional as well as visual: a door without a key breaks a level whereas broken symmetry in a GAN-generated image is often merely odd without deviating far enough to hinder the use of the image. They emphasize the earlier distinction from search-based PCG, pointing out that when PCG games include optional content the generator can be allowed to produce failed results as long as enough of them are good: e.g., the player is expected to throw away a lot of sub-par guns in *Borderlands* [212].

There's an additional distinction here that the paper elides. An artifact can be *sub-par but playable* (a gun with bad stats in *Borderlands* [212]; a level in *Crypt of the NecroDancer* [116] that you use the shovel on to compensate for an undesirably difficult dungeon floorplan) but we can posit a second category of failure: generating artifacts that are broken to the point of making the game *unplayable*. For example, a level-generation GAN that fails to include a traversable path to the level exit. There are strategies to mitigate this, including filtering them out (Sec. 8.1.3), adding design elements that turn them into the sub-optimal-failure category above (e.g. the *Necro-Dancer* shovel), and so on, but it's useful to recognize that there's a difference between *sub-par* and *broken* artifacts that needs to be considered when designing the system.

Sub-par artifacts can often be intentional, similar to how "weak cards are a fundamental part" of collectable card games [295]. The mushroom quality of sub-par guns in *Borderlands*—to use Emily Short's terminology (Sec. 3.9)—are a part of the game's economy, making the unique weapons stand in contrast (Sec. 5.5.2). Sub-par results still exist within the frame of the game. Broken artifacts, on the other hand, are broken on a meta-level: the artifacts cause problems for the context that they are supposed to work within.[8]

The focus on functionality here is probably attributable to the fact that deep learning is much harder to control than an approach such as using a correct-by-construction grammar. The mapping between the latent space of the deep-learning neural network

[8]The dividing line between broken and sub-par is, admittedly, colored by player perception: "Interesting that this distinction is at least sometimes player-specific, e.g. two players at different skill levels (or even different points in the metaprogression) might describe the same Hades encounter as 'unplayably difficult' vs 'a walk in the park'. Some failures, however, are total (e.g. the 'no path to exit' example in a game where you can't carve new paths) whereas some are gradient. And in the gradient-failure case (or in total-failure cases that look like gradient-failure cases, e.g. where a level is somehow numerically unbeatable due to emergent properties of its design but this isn't obvious without doing a lot of math), there'll probably always be some player sufficiently skilled/masochistic enough to take overcoming these generator failures as a challenge." (From a discussion between Nic Junius and Max Kreminski) [197]

and the desired generative space is not always clear or intuitive.

Liu et al. identify five broad deep learning methods in the PCG literature: supervised learning, standard unsupervised learning, reinforcement learning, adversarial learning, and evolutionary computation. Supervised learning is used “often as a predictor” to evaluate content quality, meet preferences, and adapt difficulty; unsupervised learning is used to sample new content from learned representations and to predict related content; reinforcement learning has been explored to maximize content quality, though this requires some adaptation, not to mention a definition of quality; the previous applications of evolutionary computation are broad enough that the category does not lend itself to a neat summary; and finally, adversarial learning is dominated by GANs and is frequently used for generating images and other arrays of tiled artifacts [209]. One further possibility for adversarial learning is cited: Bontrager and Togelius’s hybrid that uses an agent to play a generated level, and then the generator uses the success of the agent as a quality metric [30]. This points towards how deep learning can act as a component that serves as a part of a larger whole.

In particular, evaluation is useful for more than just a loop to refine one generator. Evaluating artifacts gives the generative system a way to read and respond to the properties of the artifacts that it is creating [200]. Or, to state it in another way, the evaluation can be used for purposes other than feeding back into the artifact-generator. For example, if we were building a Dwarf-Fortress-like, and our artifact is a complex, multi-dimensional agent-based simulation of a society, we can use deep learning to classify it into a particular architype. Other parts of the generative system can use that classification to influence their choices: e.g., this part of the latent space is classified as an egalitarian society, so the dance generator will tend to prefer dances that involve ensembles instead of prima donnas.

This is something that Liu et al. acknowledge, though they consider it one of three indirect uses of deep learning for content generation: evaluating game content, construction of human-player-imitating bots, and modeling player experience [209]. Additionally, they list a few of the existing applications for deep learning in PCG: “deep learning may serve as a content generator, as a content evaluator, as a gameplay outcome predictor, as a driver of search, and as a pattern recognizer for repair and style transfer” [209]. However, a more general picture appears if we consider each of the indirect uses of deep learning to be one of many possible ways to configure a system of evaluation and generation nodes, particularly if we consider that mixed-initiative player-in-the-loop generation can include the player as just another node. This makes it easier to see how the exact same deep learning evaluator can be deployed as a part of many different generative systems.

Looking forward, Liu et al. discuss some of the areas that are particularly important “for the current and future use of DLPCG in games”: “mixed-initiative generation, style transfer and breeding, underexplored content types, learning from small datasets, orchestrating different content types within a game, and generalizing generation across games” [209].

3.5 Understanding Procedural Content Generation

Gillian Smith's "Understanding procedural content generation: a design-centric analysis of the role of PCG in games" presents a framework "created by analyzing several PCG games and research projects through the lens of the Mechanics, Dynamics, and Aesthetics (MDA) framework" [323]. One aspect of PCG that the framework is particularly concerned with is *replayability*, arguing that a more nuanced understanding is necessary.

The framework lists five categories for the approaches used (optimization, constraint satisfaction, grammars, content selection, and constructive). The mechanics-level aspects of a generator are classified as **building blocks** (experiential chunks, templates, components, subcomponents), **game stage** (offline, online), **interaction type** (none, parameterized, preference, direct manipulation), and **player experience** (indirect, compositional, experiential). Dynamics-level aspects are in relationship to other mechanics (core, partial framing, decorative), memorization vs. reaction (memorization, reaction), strategizing, searching, practicing, and interacting. And finally the Aesthetics (in the somewhat misleading MDA sense of Aesthetics [165]) are discovery, challenge, and fellowship.

These divisions point out useful distinctions, and separating them out into different concerns is very clarifying. The poetics of a generator (Chap. 5) and the construction of a generator (Chap. 8) have a cause-and-effect relationship, but they are clearly different categories.

3.6 Generative Methods

The confusion produced by the many different terms used to describe generative systems has gotten in the way of cross-discipline research. One attempt to rectify this is Compton, Osborn, and Mateas' "Generative Methods" [66]. After a discussion of the difficulty of finding a consistent definition of "Procedural Content Generation", they propose instead to substitute "Generative Methods" which they define as "*methods* which *generate* some artifact" and "Abstractly, a generative method is a function which produces artifacts" [66]. An artifact, in their definition, "is something which contains non-trivial structure (an interior) and a context in which it is used (an exterior)" [66]. This definition draws on Herbert Simon's terminology in *The Sciences of the Artificial.* They position the sine function as a non-generative-method, since it has "no inherent context or purpose" [66].

This is, in my view, a recapitulation of the monolithic mistake: the modules that make up a generative system frequently lack an exterior (e.g. Perlin noise) and it is the generative system *as a whole* that supplies the exterior (cf. Sec. 8.6). The difference between the data being passed through and transformed by the system and the final artifact is, in practice, a difference in use rather than a difference in kind.

The definition of "content" is, once again, a hazard. In attempting to avoid the complication of what counts as "content" they nevertheless attempt to make the exact same fuzzy distinction between simple and complex, casting the sine function as being even less than mere generation.

While “Generative Methods” does discuss generative methods that are themselves composed of generative methods, by demanding that *every* generative method have a comprehensible exterior we are denied access to the interior operations. By disambiguating between *generative systems* and *generative operations* we can construct a much clearer picture of how they operate. “Generative Methods” describes generation as a cell, with a generator acting as nucleus. In contrast, I argue that a *generative operation* is closer to a neuron: an individual component but tightly interlinked with other operations. A generative system, therefore, is like a nervous system, receiving sensory input and generating signals in response.

Generative *operations* are re-usable, composable, and may or may not have mimetic context associated with them.

Nevertheless, “Generative Methods” does make an important distinction of purpose between the *assets*, input values; the *generator*, which uses the inputs to create artifacts; and the *critic* which accepts or rejects the artifacts. This is a very reasonable anatomy of many generative systems, as excellently illustrated by the figures in “Generative Methods.” However, this can be an oversimplification of the possible arrangements of the metaphorical body parts. Instead of thinking of a critic as part of a generative method, it is more straightforward to view it as a generative operation in its own right: there’s no reason it can’t receive input from multiple generators, or have its evaluation function set by parameters or even generated by yet another generator. We can generalize a critic to be a kind of filtering operation (cf. Sec. 8.1.3).

One very illuminating move that “Generative Methods” proposes is that a human can act as a critic in a generative method, giving several examples of this in action. However, we can go further: we posit below that a human can take the role of *any* operation in a generative system (cf. Sec. 8.4). For example, crowdsourcing Foldit puzzle solutions [78] arguably uses humans as generators and the computer as the critic. This is, admittedly, complicated by the interface problem: data must be translated into a human-comprehensible format and then the human input in turn needs to be translated into a machine-comprehensible format, using a form of *generativist reading* (Chap. 6).

A further important contribution in "Generative Methods" is connecting with generative research from other non-game fields. Comparisons across generators from different fields can illuminate the commonalities and help reveal which elements are atomic. Identifying common threads can "help us see a broader and more interesting set of candidate systems for analysis and synthesis" letting us compare topologies, algorithms, and strategies to better understand the "distinct set of affordances" in each system [66].

3.7 Generative Framework

The framework that had the most influence on this present work is "A Generative Framework of Generativity" [64]. Beginning with the observation that many fields use the same generative methods to different ends, the framework presents a way of thinking about generativity as a pipeline of data transformations. This parallels the implicit wild taxonomies, where the stack of modifiers or web of nodes are modularly built up into a complete generative system, rather than the generator being a bespoke oracular monolith.

Compton et al. present eight broad categories of transformations that can make up parts of the pipeline, and three kinds of consumers that consume entire pipelines. The transformation categories are content selection, tiles, grammars, parametric, geometric transformations, distributions, agent-based, and machine-learned statistical models. The three "consumers of finished pipelines" identified are search-based techniques, constraint solvers, and human interactivity [64].

This framework was further refined with "Generominos: Ideation Cards for Interactive Generativity" in which Compton et al. present a physical representation of the framework in the form of ideation cards, which can be arranged to represent the dataflow through the generative pipeline. This concrete reification of the framework further refines the model, incorporating scenario cards to encourage thinking about context and being more explicit about what input and output data types are similar. Outside of its use as a theoretical taxonomy, Generominos have been used for project planning, reifying theories of interactivity, and as part of educational instruction [65].

An important part of the Generominos framework is a focus on data types: a transformation has a defined input type and can only receive information of that type. This allows for the deeper analysis of generative and interactive systems: being able to point out that devices that produce input other than events or states, such as the Kinect, "can only plug into standard game designs after lossy or indirect transformation of their input data to events" [65]. This insight highlights the importance of including data types in a conceptual framework of generative systems. It also reminds us that a framework should be useful for concrete analysis. The Generominos framework both indicates the problem (lossy transformation of input) and suggests what design changes would need to be made to compensate (e.g. the direct joint control of Compton's *Idle Hands* [58]).

There are, however, areas where this framework can be improved upon. The treatment of search-based techniques and constraint solvers as "consumers of generative systems" elides the details of how their output can in turn be integrated into a generative system. Generominos are excellent as ideation cards, but implementing the system that you design with them requires additional engineering effort to successfully reify the entire pipeline. Using the Generative Framework as a starting point, we can improve on it, rethinking the details gives us a framework that can better describe complete procgen systems in context (Chap. 8)

Compton's more extensive description of this framework, in Chapter 7 of [59], is worth consulting, especially because of the more extensive cataloging of the different categories of generative methods.

3.8 A Framework for Understanding Generative Art

Elsewhere, Compton discusses generative art frameworks, singling out "A Framework for Understanding Generative Art" by Dorin et al. [89] as a major influence on her generative framework [59, p.98].

Dorin et al. describe generative art in terms of four components: Entities, subjects on which the system acts; Processes, “the mechanisms of change that occur within a generative system”; Environmental Interaction, involving the “flows of information between the generative processes and their operating environment”; and Sensory Outcomes, “the experienced aspects of a generative work” [89].

Compton modifies the framework by emphasizing that the system must form a chain from input to output, pairing this with more separation between the interior and exterior of the generator and eliding the entities within the generator [59, p.100]. Most critically for our purposes, Compton draws a clear separation between the generator and “how people interact with, experience, and explore it” [59, p.100], a vital move that cuts through the confusion in many of the previous frameworks.

One other aspect of the framework by Dorin et al. that was implicitly continued by Compton is the idea of mapping “the transformation of the underlying entities and processes of the system to perceptible outcomes,” so that the obscured processes of the generative system can be “rendered perceptible” [89, p.12]. While some systems are directly-perceivable and therefore described as *flat*, there are also systems with *natural mapping* with close alignment between “entities, processes and outcome” and therefore an alignment in ontology [89, p.12]. Other systems, such as cellular automata, allow for *arbitrary mapping.* (Compare with the disussion of teleological and ontogenic modeling in Sec. 6.2.2.)

In a more general sense of mapping, we can construct a mapping between the “Understanding Generative Art” framework and my framework (cf. Sec. 8). In my terms, the entities are artifacts, but with an emphasis on how they are also the data being processed. This union especially makes sense in a generative art context, where many works are about acting on a specific, singular medium or subject. In a wider procgen context this highlights how any part of the data passing through the system is a potential artifact, though this is often obscured by how small bits of data (two numbers, for example) are meaningless without context (e.g. the position of a star). Processes are transformations (and the term is apt: process might be a better description of their role). Sensory outcomes are one kind of output sink, and environmental interaction is either an input source (Sec. 8.1.1), or an interaction loop that goes outside the formal system and then returns.

3.9 Annals of the Parrigues

As we will see in Chapter 5, there is another, very different framework that has seen use in practice: Emily Short's aesthetic tarot principles. Introduced in the afterward to her 2015 collaboratively-authored-with-the-computer travel guide generated novel *Annals of the Parrigues* [310], Emily Short' five principles were originally a set of world-building partially-opposed iconographic poles that eventually developed into the fabric of the generator itself. The five principles are named, somewhat poetically: Salt, Mushroom, Beeswax, Venom, and Egg.

Salt is about elegant order and grammars. "For the principle of salt, the machine-that-writes matters more than the thing-written" [310, p. 88]. Beeswax is specific rather than generalized, uses one-off elements, and sources corpora from things other people have hand-assembled [310, p. 90]. "The Principle of Venom recommends that we focus our variational efforts on the most statistically implausible, meaning-bearing words in the sentence" [310, p. 94]. Egg allows for human curation and selection: "The egg principle is the principle of consensus, the principle of combination, or the principle of the authorial self" [310, p. 98].

Lastly, Mushroom is the repetitious fungus on the forest floor of procedurality. "Mushroom-writing does not care about an individual instance of output and does not regret the loss of any element" [310, p. 89]. Short thinks of "Markov chains as mushroomy" [310, p. 89]. Though not mentioned by Short (who is focused on text generation) another example of something frequently used for mushroom aesthetics is Perlin noise, particularly when used for basic terrain generation.

These are aesthetic principles, and their poetic definitions reflect that. Nevertheless, Short incorporated them into both the content and technical aspects of her generative novel. Short's five principles were not intended as a universal framework: they are intertwined with the original travel guide's topics. Nevertheless, other developers have found them helpful and developed systems based on them, such as the system Bruno Dias used for the game *Voyageur* [85–87].

Therefore, other aesthetic tarots are possible. However, we can still use these particular five principles as an aesthetic lens for our vivisection. For example, the opposition between the repetition of Mushroom and the significant variation of Venom. Often when constructing a generative system, we want its emergent expressive range to have landmarks: artifacts that stand out so much that the player notices them and uses them as a way to orient themselves in the generative space of possible artifacts. We could design a generative system that only produces landmarks: every possible artifact stands out from the rest. However, it is usually more effective to take a Mushroom approach and have a large pool of repetitious artifacts with minor variations so that the few emergent Venomous artifacts stand out as landmarks.

3.10 The Topology of Generative Systems

By now it should be apparent that the *operations* in a generative system are only part of the picture. How the operations are connected—the topology of the generative system—is equally important when analyzing what the generative system is doing.

A framework modeling procedural generation should be able to take into account that the exact same operation can be deployed in many different uses (cf. 8.6). For example, Perlin noise [267, 269] can be used to generate wood, marble, and other textures [267]; simulate smoke and fluids [41]; create a heightmap for a landscape [294]; describe the density of a cloud [176, p. 39]; drive an animated character [268]; simulate corrosive metal for image processing for rust detection [2]; synthesize audio [275]; perturb music generation [372]; and countless other applications. Even reducing Perlin noise to primitive input and output types is complicated by how it can be combined with other operations to produce a scalar, a vector, a flowfield array of vectors, or a scalar by tracing a complex path through the vector space. Perlin noise is a versatile building block that can be used in many different ways. Generative systems can't simply be characterized by what generative operations they use. How those operations are connected is just as important.

Further complicating the picture is that generative systems can be combined with other generative systems. The most obvious manifestation of this is the fact that generative systems can contain nested generative systems as operations, which has gotten a fair amount of attention. However, the recursive relationship goes deeper: generative systems can themselves generate entire generative systems–and this is a regular use of generative systems in the wild. Generative systems can themselves be generative artifacts: the "something" in "make something that makes something" [72] can itself be a generative system.

3.11 Conclusion

I hope this chapter makes it clear that there is a significant divide between how we theorize about procgen in a research context and how we actually model procgen when we build tools to facilitate its practice. Some theoretical frameworks come much closer than others (particularly the pipelines modeled by Generominos) but in general the gap between practice and theory is a wide ravine.

A major theme in this chapter is perspective: many of the frameworks discussed here are useful in their original context (or part of their original context) but highly misleading in others. This very much depends on the question that you are trying to answer when you use one of these frameworks. Applying a framework to the wrong question leads to category errors.

When I sit down with a student and try to use a framework as a checklist for understanding what they are trying to build, I get very mixed results with many of these frameworks: from this perspective, many of them are fundamentally geared towards answering the wrong questions. Asking as matter of first principles if my generator is online versus offline, for example, is in my experience almost always premature optimization: later performance constraints may alter my design choices. In my experience a more useful starting point is to ask questions about what it is we want the generator to do: What effect should it have on us (Chap. 5)? How can we write rules to construct this particular artifact (Chap. 6)? What are the individual pieces we need to assemble to construct the generative system that we want (Chap. 8)?

Chapter 4

Generators in the Wild

In addition to the frameworks that describe procgen, there are the generators themselves. This chapter contrasts the formal and explicit frameworks that we have used to analyze and describe generators and asks how closely they align with how generators are constructed in practice.

There are, fortunately, many different examples to draw on. And we don't have to confine ourselves to individual generative systems. There are numerous software tools that function as generative system construction tools. Each tool implicitly models generative systems in order to facilitate constructing a generator for their particular domain. Therefore, the models of generativity that are most widely known, at least among these active communities, are these implicit theories of what a generator needs.

Drawing on methods from software studies, this chapter dives into many examples of the tools that exist and how their designs can inform our understanding of procgen. We can divide them into two broad categories: generative system construction tools that use a modifier stack, and tools that use a node-based paradigm.

4.1 Modifier Stacks

The simplest and most direct generative system is a waterfall of generative operations, with each operation feeding into the next. In Autodesk's 3ds Max [14] this has been reified as a "modifier stack," a list of non-destructive operations that are performed in sequence to transform a 3D model [13].

These *Modifiers* are operations that include: the object's creation parameters; twisting or bending the object's geometry; applying materials and UV unwrapping; selecting part of an object (which other modifiers can reference); adding skinning weights for rigging; extruding or lathing an object or part of an object; and quite a few more. We can view each of these modifiers as a *data transformation*: a modifier takes a 3D object and its associated data, and returns a newly-changed 3D object.

The 3ds Max terminology for the order of data transformations is "object data flow" [12] of which the modifier stack is one step in the object data flow. The modifier stack can also be collapsed, converting the live generative system into a fixed artifact. In the case of 3ds Max, the fixed artifact is typically an "editable version of the original object" [11] in the sense that the user can directly edit the vertices or polygons of the 3D object, rather than indirectly affecting them via the modifiers.

Other 3D modeling programs, such as Blender, also incorporate a modifier stack:

"Modifiers are a series of non-destructive operations which can be applied

> on top of an object's geometry. They can be applied in just about any order the user chooses. This kind of functionality is often referred to as a "modifier stack" and is also found in several other 3D applications." [27]

The modifier stack passes down a rich and monolithic representation of the 3D object: necessary, because each generative operation can only work with the context it receives from its most immediate predecessor. Therefore, the limitation of a modifier stack is that it is generally specialized to one common (and often complex) data type.

While the modifiers in 3ds Max can include additional data (animation controllers, instancing, multi-object modifier stacks, scripts, sub-object selection volumes, etc.) the modifier stack is based on the reasonable assumption that the primary purpose is generating a single 3D object, with vertices and polygons (or NURBS or splines.) Out of the box, it is not really set up to, say, combine vertices with music, limiting the context that the generative system can make use of.

While there are ways to represent additional input data, modifier stacks are most effective when oriented around a single kind of artifact, which is then constructed via a linear waterfall of transformations.

4.2 Node-Based

A more common way to represent generative systems in the tools used to build them is as a *node-based* network. These can be found in software across many different creative fields, from music to video compositing to generative art to 3D graphics. They are closely akin to visual programming languages (VPLs) and as such often implement ideas from data-flow programming (see 4.3).

One distinction between node-based graphs and a flowchart of software execution is that these generative node-based systems tend to depict the flow of *data* rather than individual execution traces. For example, while shaders are often designed from input-to-output (left-to-right or top-to-bottom in many visualizations) the compilation frequently assembles the end result by starting with what the output needs and works back through the graph (essentially placing the outmost loop on the right or bottom).

4.2.1 Video Compositing

There are two broad approaches to film compositing: layer-based compositing and node-based compositing [224]. These mirror the distinction between a modifier stack and a node graph.

Layer-based compositing, such as found in software like Adobe's After Effects [3], has the different elements arranged in a stack of layers, similar to the layers in Adobe's Photoshop [4], or to the modifier stack in 3ds Max. However, for complex compositing with hundreds of interlinked elements, a layer-based workflow requires sub-compositing and nesting [224].

A node-based compositing approach, such as used by The Foundry's Nuke [350], instead treats each operation as a separate node in a graph of data transformations. Each node has inputs and an output, with "pipes" connecting the output of one node to one or more other nodes. The flexibility of this approach allows one data operation to be reused in multiple places.

4.2.2 Textures and Shaders

In 3D graphics, texture shaders describe the way that a 3D surface is rendered, with the color, reflectivity, and other aspects of the visual appearance specified for each pixel through what is essentially a small generative program. These can be written programs (e.g. HLSL script) but are often represented visually as graphs of nodes. This is widespread across current 3D graphics software (including 3ds Max, Maya, Lightwave 3D, Cinema 4D, Houdini, and Blender.)

One of the most well-known node-based shader-construction software packages is Substance Designer (now owned by Adobe) [6]. Substance Designer is a "node based texturing software" where the user constructs shaders out of nodes and compound graphs of nodes [5]. This reuse of graphs of nodes is fairly common across node-based generator-system-building software, allowing large libraries of generators and sub-generators to be collected. In the case of Substance, the company runs a marketplace for node graphs[1], has a 3D painting program that uses the node-based shaders for texturing, has a separate software package for collecting and tweaking libraries of node graphs (Substance Alchemist), and offers plugins that allow the shader node graphs to be used directly by 3D software like Houdini (Sec. 4.2.2.3) and game engines such as Unreal and Unity.

In the music world, Pure Data [280] is a modular composition program that acts as a dataflow programming environment (Fig. 4.1). In Pure Data, the document the user edits is a *patch*, which consists of "a collection of boxes connected in a network,"

[1] (`https://source.substance3d.com/`)

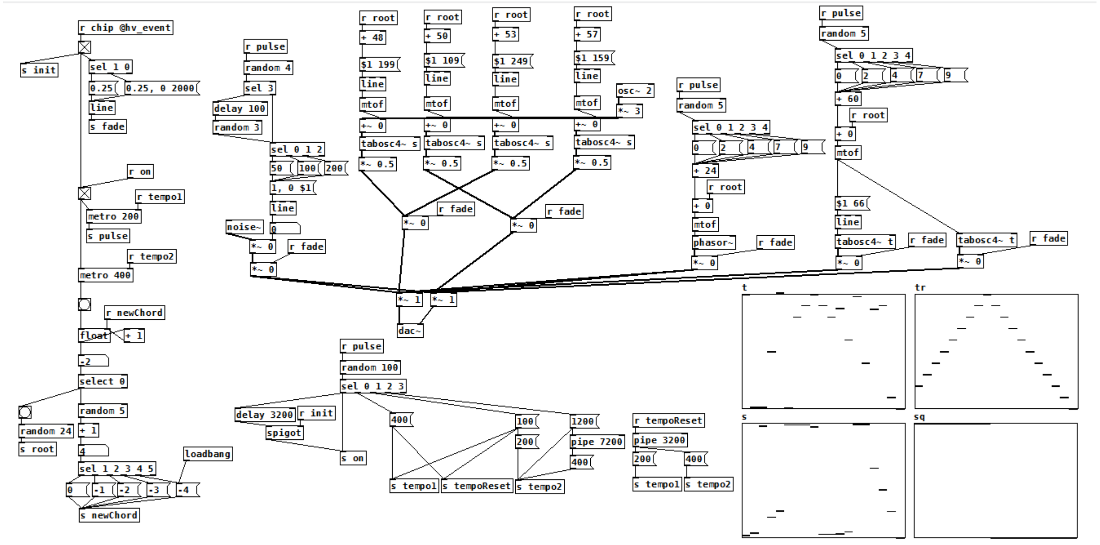
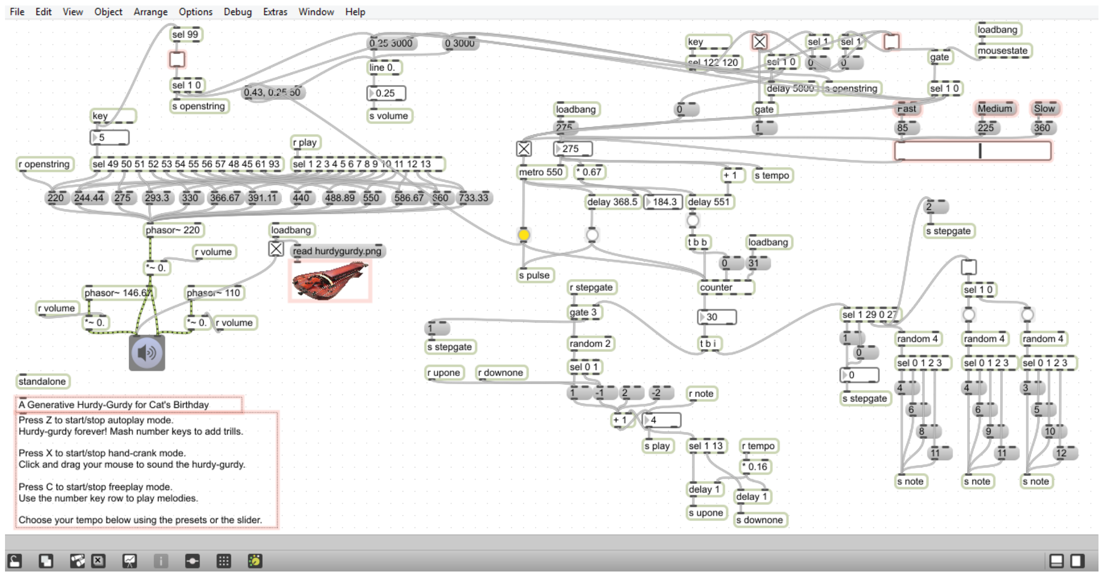


Figure 4.1: *InstantChip* in Pure Data, by Tamara Duplantis. Parses randomized rhythmic and harmonic logic into music. The on switch is in the top left corner, controlling the fade in/fade out and the metronome. The metronome's periodic pulses trigger the instruments (on the upper row). The bottom row function has a low probability chance to change major characteristics of the tempo, to "allow for gradual variety over long listening sessions" (Tamara Duplantis, personal communication) Image courtesy of Tamara Duplantis, used with permission.

in three categories: message boxes (which send a message when activated); object boxes (which specify a process to enact when interpreting the messages it receives); and number boxes [279, p. 15]. Pure Data also makes a distinction between discrete messages and signals (which are continuous streams of audio data).

Another music dataflow environment is Max [1] which was originally also created by Miller Puckette–the same developer who would later create Pure Data. Max similarly acts as a visual programming environment for dataflow programming (Fig. 4.2).


File Edit View Object Arrange Options Debug Extras Window Help
Fast
Medium
Slow
standalone

A Generative Hurdy-Gurdy for Cat's Birthday

Press Z to start/stop autoplay mode.
Hurdy-gurdy forever! Mash number keys to add trills.

Press X to start/stop hand-crank mode.
Click and drag your mouse to sound the hurdy-gurdy.

Press C to start/stop freeplay mode.
Use the number key row to play melodies.

Choose your tempo below using the presets or the slider.

Figure 4.2: *A Generative Hurdy-Gurdy* in Max 6, by Tamara Duplantis. The instruments are on the left: three sawtooth tones that are analogous to the strings of a hurdy-gurdy, connected to the digital audio converter "easily identified as the big speaker object," with logic in the center for the metronome-driven beat (Tamara Duplantis, personal communication). Image courtesy of Tamara Duplantis, used with permission.

4.2.2.1 vvvv

Overlapping partially with music, live coding environments, such as *vvvv* (pronounced “v4”) are also intended for creating real-time motion graphics together with audio. Like many other datflow environments, vvvv uses a node-based approach for its visual editor (Fig. 4.3).

4.2.2.2 werkkzeug

From the demoscene, werkkzeug is a tool by demoscene group .theprodukkt, for 64kb demos, who also used it in projects like their 96kb game *.kkrieger* [258]. One primary distinction between werkkzeug and other visual programming environments is that the nodes are displayed as blocks, with vertical coincidence designating input and output from each block (Fig. 4.4).

4.2.2.3 Houdini

“Nodes are the basis for everything that Houdini does” [315]

SideFx’s Houdini [314] makes nodes the central organizing feature of the 3D graphics software (Fig. 4.5). User operations that would be destructive operations in other software are instead converted to nodes behind the scenes, and can therefore be edited and rearranged later. This keeps the generative system live for more of the production process, in contrast to the tendency of other 3D software to start from an assumption of a fixed artifact that has operations applied to it and is finally baked back into a new fixed artifact.

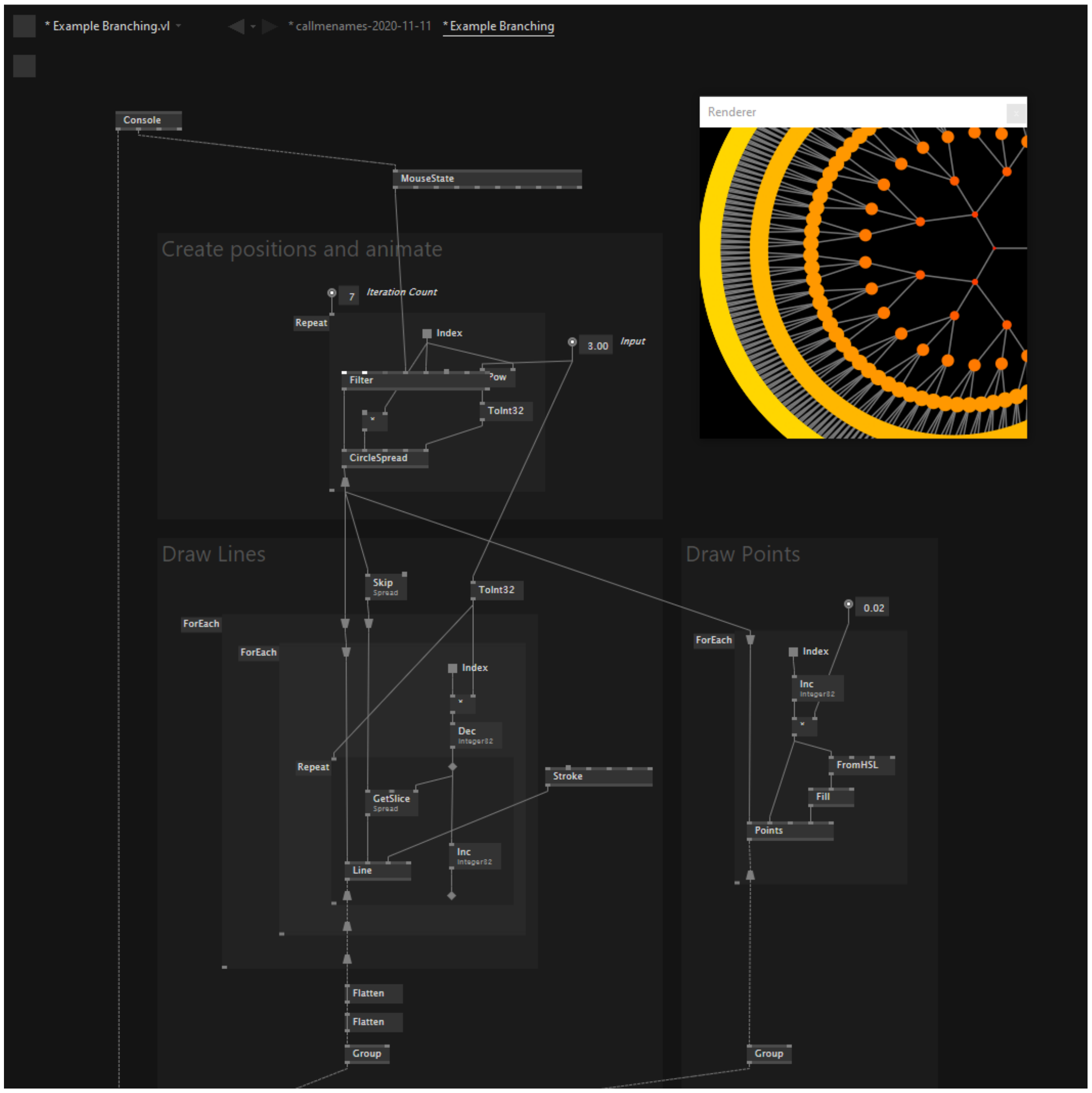


Figure 4.3: The node-based interface of *vvvv gamma*.

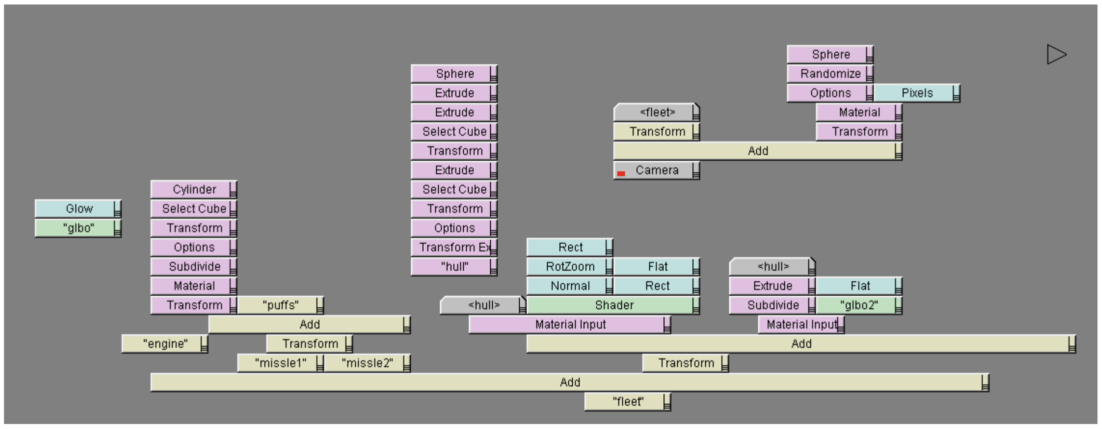


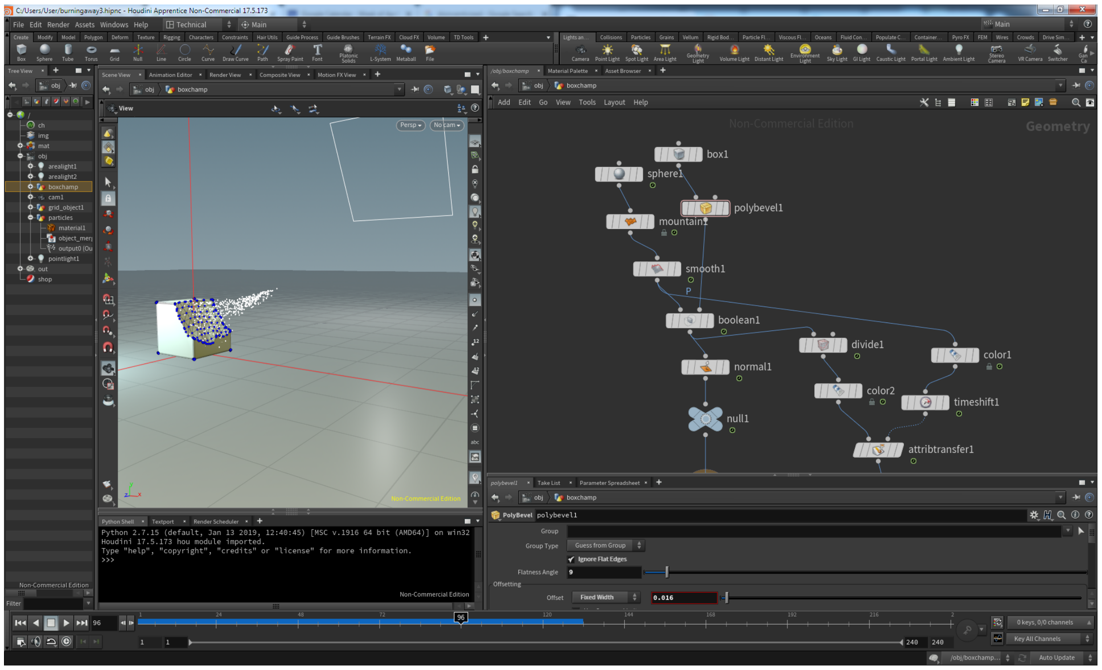
Figure 4.4: Screenshot of werkkzeug. Note that the relationships are expressed by the neighboring blocks rather than visualized with lines, which is an intentional design decision to keep visual programming in werkkzeug more manageable [258]. Image courtesy of Adam M. Smith, used with permission.

The system being live also means that non-visual elements can be incorporated into the node networks. For example, take Andrew Lowell's talk on using geometry as a visual programming language, where he takes advantage of the capability of attaching arbitrary data to points in space to use a physics system to drive a music composition [216].

4.3 Dataflow Programming

As you might have noticed, there is a relationship between software for constructing generative systems in the wild and visual programming languages. While there are certainly non-visual systems for constructing generative systems (notably graphics shader languages) the node-based generative systems predominate.

Node-based generators are often based on dataflow programming (also referred



```
Python 2.7.15 (default, Jan 13 2019, 12:40:45) [MSC v.1916 64 bit (AMD64)] on win32
Houdini 17.5.173 hou module imported.
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 4.5: The node-based interface of Houdini. All of the operations in Houdini are represented as nodes, displayed here on the right.

to as “data flow programming” or “data-flow programming”). In dataflow programming, programs are represented internally as a directed graph [335]. The graph is composed of blocks (or nodes) that act as sources, sinks, or processing blocks, connected “by directed edges that define the flow of information between them” [335]. Dataflow programming’s strengths include the ease of visual programming and concurrency.

Dataflow Programming’s origins can be traced back to Sutherland’s SKETCHPAD [335, 347] and influenced most visual programming languages (VPLs) [335]. Because the nodes are essentially pure functions without side-effects, they can be easily parallelized.

While many generative processes can be represented as acyclic graphs, some operations (such as a generate-and-test loop) introduce cycles into the graph. The best way to represent cycles and iteration in a dataflow programming language is an ongoing research topic but several implementation approaches exist [173]. In a survey of visual programming languages by Mosconi and Porta, the authors identify five approaches, using the VPLs Show-and-Tell, LabView, Prograph, Cantata, and VEE as representative languages, as well as investigate the minimum set of characteristics for implementing iterative behaviors in a pure data-flow language [242].

Concurrency in dataflow programming can be implemented, among other methods, by representing operations as activity templates [84] or by using a Khan Process Network, where the nodes communicate via unbounded FIFO input queues [128]. This is significant for generative systems because the relationship between semantic positioning and execution order can be non-obvious. Analysis of a generative artifact often is best performed by starting from the generative artifact and working backwards through the inputs of the generative system that created it.

Chapter 5

Procgen Poetics

How does the structure of a generative system impact the aesthetic effect of the output? (RQ1) This question is often the entire point about talking about generative systems in most current applications. We have a generative system that we are using for artistic ends—either as a generator in itself, such as with a level generator, or as a component of something else that has an aesthetic motivation. Even in industrial design, a generative design frequently has to interact with the space of human experience.

Games in particular have this as a primary question. Which leads to the problem: while Procedural Content Generation (PCG) is an essential element of many games—including analog games [324]—and generative methods in general have a wide variety of applications, there has been a lack of vocabulary to discuss the effects that different generative systems have.

This is also one of the gaps between the academic discussions and the practitioners: without methods of critique, we cannot hope to have a useful dialog. The lack of critical discourse around procgen is, in large part, because we don't have widely accepted ways of talking about what the effects of a given generative system are. And, in turn, practitioners have more difficulty understanding how a newly-researched algorithm fits into the systems they are trying to construct.

We need a way to talk about the connection between how a generative system works and the effects that it has on us. One way is to build on something I have previously proposed: describing a poetics of procgen [183].

Poetics is a method of modeling literary effects by describing the elements they arise from [79, p.69]; a poetics, therefore, is a model of the poetics of a specific domain. For a poetics of procgen, we can start by studying the effects we observe in generative systems and ask how the system creates those effects [79, p.61]. What part of the generative system in front of us leads to it having this specific effect on us? What kinds of effects can procgen systems have? And, crucially, what kinds of effects are difficult or impossible for a particular approach to procgen, or for procgen in general?

There are many different reasons for using PCG. The effect that a particular work is trying to create is dependent on the developers' motivation. Likewise the effect that two works have in practice can be very different even if they look similar at first glance. Even if we look at some of the earliest major touchstones for PCG, they have very different motivations behind them. For *Elite* [38], the motivation was to fit the galaxy in the very limited memory of the BBC micro:

> Their first idea had been to furnish the machine with the details of (say) 10 solar systems they'd lovingly handcrafted in advance: elegant stars, advantageously distributed, orbited by nice planets in salubrious locations, inhabited

> by contrasting aliens with varied governments and interesting commodities to trade. But it quickly became clear that the wodge of data involved was going to make an impossible demand on memory. [336]

For *Rogue* [357], on the other hand, the motivation was to surprise players, including the creators:

> One of the things we wanted to do was create a game we could enjoy playing ourselves. Most of the existing adventure-type games had "canned" adventures – they were exactly the same every time you played, and of course the programmers had to invent all of the puzzles, and therefore would always know how to beat the game. We decided that with Rogue, the program itself should "build the dungeon," giving you a new adventure every time you played, and making it possible for even the creators to be surprised by the game. [375]

While *Elite* has a similar sense of hands-off surprise, its fixed galaxies mean that ultimately its procedural generation is attempting a different effect. Criticism and analysis of generative systems needs to be able to take into account that there are different purposes for using generative methods. Different systems have different aesthetic effects, and what might be a good generative system in one context may not be fit for purpose in another. But we can't talk about this without the words to express it. We lack a poetics of PCG.

This chapter is partially motivated by Mike Cook's call for interrogating the language we use to talk about procedural generation. According to him:

> The old language of procedural generation needs to be done away with, and in its stead we need a new way of communicating about what we do, and why it's interesting. We need to debunk the idea of procedural generation as a dark art, and show people that it is accessible, understandable and interesting. [73]

Marketing for these games uses “words like discover, unique, endless, forever, replayable” and implies that “bigger is always better” [73].

To that end, this chapter presents vocabulary that we can use to describe the poetics of procedural generation—the ways in which generativity communicates meaning. Instead of talking about big numbers, or quadrillions of planets,[1] we can instead talk about the effects that the generative system invokes in the player. Whether that is through intentionally overwhelming the player with the sheer size of the generative space; the player’s attachment to a particular, unique generated artifact; the creation of a sense of ritualistic repetition; or through some other aesthetic effect.

Poetics, as used in this chapter, refers to a theory of form that studies the creative principles informing a creative work. You can, if you like, take it as a set of aesthetic aspects that combine to give us a model of what effects a generative system can have on us.

Just as “Towards a Theory of Choice Poetics” (Mawhorter et al., 2014) describes a poetics of choices in games, in this chapter I present a theory of generative forms. While taxonomies of procedural generation do exist, the focus has been on the *how* rather than the *why*. Many taxonomies of procedural generation, such as those presented in Togelius et al. (2011), Yannakakis et al. (2011) and Hendrix et al. (2011) have been oriented around either the domain where the generation is applied or the spe-

[1]For example, the marketing for *No Man’s Sky* emphasized that it uses 64-bit seed to generate a potential 2^{64} planets [156], but that doesn’t give us the actual expressive range. In contrast, the generative system in *Elite* is capable of generating 2^{48} *galaxies*, but the publisher insisted that they limit it to eight to preserve the player’s interest: “Somewhere between the unimpressed response to a small game universe and the disbelieving response to a ridiculously large one lay a zone of awe” [336]. Fortunately, the actual post-release development of *No Man’s Sky* has focused on making the planets more expressive and giving the players more to do with them.

cific techniques that the generation uses. The poetics of generativity—what it means when we use a particular form of generation and what effect it has on the player—are under-explored. This is unfortunate because understanding the poetics of procedural generation not only points out new directions for future generative research, but also better equips designers to make decisions about how and when to use generative systems, *and* gives critics the vocabulary to properly dissect the systems they are critiquing.

Therefore, this chapter describes an understanding of the poetics of generative content in games. After discussing previous frameworks, the chapter is structured as a series of lenses, each of which brings focus to a set of aesthetic dimensions (Table 5.1). The relationships between aspects of the lenses are described in spatial terms, metaphorically situating them in space, in an analogy with the way that generative forms deal with distributions in abstract, mathematical probability space. This *generative possibility space* is an aesthetic element that separates generative forms from other media.[2]

Like the visual vocabulary of montage in film or the use of meter and metaphor in language, the poetics of generative systems are built out of this vocabulary of forms.

Many existing aesthetic theories deal with composition, such as composition in time or in the picture plane. Some media vary across space, such as the panel composition in comics [227]. Other media vary in the more abstract dimension of time, such as cuts in film [202, p.5] or measures in music.

What sets generative art apart from other media is that it is composed in a

[2]For example, the game Blabrecs is partially about exploring its "shadow English" possibility space. [198]

completely different dimension: parametric probability space. By imagining the possible results of a generative system in this mathematical space, we can apply aesthetic concepts of contrast and balance to probability: for example, the bell curve from rolling a pair of dice on an encounter table is different than rolling a single die.

Where previous discussions of generative art have reduced this distribution of parametric probability to single linear continuum, *Apollonian* order and *Dionysian* noise use information complexity to describe a two-dimensional space of possible **Generative Complexity**.

The lens of **Generative Form** is about the form the generative system itself expresses: through highly-distinct *Individual* artifacts, a *Gestalt* sense of the entire output, or highlighting the relationships between artifacts through *Repetition* of the same form.

Generative Locus inverts this lens and considers generativity from the perspective of the generative system's relationship with the player. Where is the focus of the generation? Is it on the *Surface*-level results of confronting a snake pit in *Spelunky* [213]? Or the rules that dictated the placement of that snake pit, i.e., the *Structure*? Or the *Gestalt* effect of how the pit fits into the level as a whole?

Finally, the lens of **Variation** gives us vocabulary to talk about the distinctions between *Multiplicity*: generating many perceptually distinct results; *Style*: generating the right thing; and *Cohesion*: generating things that agree with each other.

Complexity	The balance between **Apollonian** (generation by rules and structures) and **Dionysian** (generation that uses chance and disorder). The combination of both produces **Complexity** and information (in the information theory sense).
Form	The balance between the **Individual** generated artifact, the **Formal Gestalt** of all the artifacts the generative system produces, and the degree of **Repetition** (or obvious lack of perceived uniqueness).
Locus	Where the player's focus is centered, in a balance between the **Structure** of the generative system's processes, the **Locus Gestalt** of the generative system's output, and the **Surface** of the immediate experience of individual generated artifacts.
Variation	The generative system's output can exhibit **Multiplicity** by generating many perceptually distinct things; **Style** by generating things that conform to an objective; or the content a generative system uses as data and how **Cohesive** the results are.

Table 5.1: A summary of the four aesthetic lenses described in this poetic framework.

5.1 Previous Frameworks

There are several prior attempts at describing an interpretive framework for procedural generation. Most of them overlap with the ideas in this framework, or are a direct inspiration for aspects of the poetics presented here. However, none of them describe a complete poetics.

5.1.1 Expressive Range Analysis

One of the more influential ways of looking at procedural generation in the research community is Gillian Smith's *expressive range analysis* [326]. "Expressive range refers to the space of potential levels that the generator is capable of creating, including how biased it is towards creating particular kinds of content in that space" [308, p.218].

Expressive range has become a common evaluation approach and has been incorporated into PCG analysis tools, such as the Danesh toolkit [76].

Measuring the expressive range of a generative system is a useful meta-heuristic to understanding a generative system [326]. However, expressive range analysis does not dictate what metrics are included in the analysis. While there has been a distressing tendency to imitate the original metrics of linearity and leniency, those metrics were intended strictly as limited examples in the specific context of Super Mario Bros.-esque linear platformer level generation [341]. There are no universal metrics: "The metrics used for any content generative system are bound to vary based on the domain that content is being generated for" [308, p.220].

From the beginning, Gillian Smith and Jim Whitehead also suggested that "These metrics should be based on global properties of the levels, and ideally should be emergent qualities from the point of view of the generator" [327]. That is, rather than measuring what the generative system's parameters directly influence, it is better to measure something indirect and emergent, which for the original use-case was the linearity of the levels generated by Tanagra [327].

While thoughtful applications of expressive range analysis have usefully considered how to incorporate it into a broader approach to PCG evaluation [341] and it has seen active use in PCG toolkits [76], there have also been an unfortunately high number of naïve misuses of the evaluation approach. A common error is over-identifying linearity and leniency with ERA when other metrics would be more appropriate. A more subtle error is to use them as metrics with a generative system that optimizes for linearity, rendering them redundant.

This highlights the difficulty of evaluating a PCG system: things that are easier to measure are also generally easier to generate—and therefore are likely to be the thing we're directly optimizing for. But for the evaluation to be meaningful it should instead be an emergent characteristic of the system. More subtly, there is no single ideal outcome for an expressive range: the different aesthetic implications of a smooth gradient across generative possibility space versus a few sharp spikes are both valid generative goals, depending on the poetic implications the generative system is designed for.

5.1.2 Gillian Smith's Design-Centric Analysis

As an approach to evaluation, expressive range is an important step in the procedural generation field, but it is not at all an attempt to construct a comprehensive framework. However, in "Understanding Procedural Content Generation," [323] Gillian Smith puts forward an analytical framework for procedural content generation, using the Mechanics-Dynamics-Aesthetics (MDA) game design framework as a starting point.

According to Smith, "It is vital that both AI researchers and designers have a common vocabulary for understanding not just what PCG is but how it can be used to induce particular experiences and what it uniquely offers to game design." The need for a common vocabulary is of particular concern to this chapter: while the field has a better grasp of the vocabulary we can use for PCG, even in 2022 there is a need for ways to talk about it that bridge the gap between practitioners in industry and researchers in academia.

Smith categorizes the approaches used in PCG into optimization, constraint satisfaction, grammars, content selection, and generation as a constructive process. While I might quibble with the details of the set, this is a good starting point for discussing some of the different major approaches.

Somewhat confusingly for our purposes, the definition of *aesthetics* in the MDA framework is misleadingly narrow: "Aesthetics describes the desirable emotional responses evoked in the player, when she interacts with the game system" [165]. This has led to some confusion over the years, as a more common definition of aesthetics would include such things as the visual style of the game, whereas the MDA framework focuses more on the more abstract sense of the machine-in-operation.[3] As this is a chapter on poetics, we need to be careful not to confuse aesthetics in this narrow sense with the concept of aesthetics more broadly.

That said, this narrow focus serves our purposes insofar as it lets us point out that the aesthetic effect of a generative system is not solely in the artifacts that are generated, but also in the relationship between them and the emergent operation of the system.

Smith categorizes the forms of MDA aesthetics used in PCG as discovery, challenge, and fellowship. These are specific "kinds of fun" discussed in the original

[3]I would characterize this as typical of the critical focus of the early 21-century, where the operation of game systems is the paramount concern because that is the aspect that is unique to videogames and therefore the most unclaimed for flag-planting by the nascent field. Many of the discussions of the time (ludology vs. narratology being perhaps the most well-known and least useful) had this as an axiomatic assumption, which had the unfortunate effect of obscuring the extent to which videogames are more usefully approached as *gesamtkunstwerk*.

MDA paper.

5.1.3 The Annals of the Parrigues

2015 was the third year for National Novel Generation Month (NaNoGenMo) [70] and the second for ProcJam [72]. Emily Short's slightly unorthodox contribution was published after the official end of the events (though both events tend to be informal about their contributions anyway) but is arguably one of the more impactful things to come out of that year's generative experiments. *Annals of the Parrigues* [311] is a travelogue novel created "in collaboration with the machine" [311, p.97]. Travelogues were a popular topic that year[4] but Short's book also incorporated a lengthy appendix with a discussion of the theory and design work that went into making it.

Building on her tarot-deck-inspired earlier attempts to generate systemic vocabularies of symbols, she settled on a set of five principles as "an organizing iconography distinct from traditional groups of elements" [311, p.86]. Importantly, the principles formed "a system of mutual partial opposition" that created a dynamic equilibrium rather than static relationships; she compares this to the use of the color wheel in the game design of *Magic: the Gathering* [311, p.86].

Short's five principles were particularly applicable to the world-building of the book she was constructing, though she also generalized them to apply to aspects of the procedural generation process itself. The principles—Mushroom, Salt, Venom, Beeswax, and Egg—were used as meta-tags on the content in the generative system's corpus: for

[4]I generated one of my own: *Virgil's Commonplace Book* [181], combining a generated traversal of the known major travel routes of the Roman empire with the texts associated with nearby locations.

example, Short associated Salt with order and regularity, so a town associated with Salt would have austere buildings in desaturated colors, and the religious beliefs would be “an organized kind of religion” [313]. Short’s idea of creating a set of principles has influenced other designers: Bruno Dias developed a similar systems of five new principles for the game *Voyageur* [87].

Additionally, and most importantly for our purposes, the principles were reflected in the design of the generative system itself: Salt is also about grammars and generative systems that operate without human input. Taken as a complete system, the five principles describe a working artist’s complete framework for a practical generative poetic, expressed in terms of opposing aesthetic forces in a deliberately dynamic equilibrium.

The framework in this chapter is, in large part, a direct response to Emily Short’s framework. However, rather than being limited to a lens focused on a single work, I take a step back and attempt to discuss aspects that her deliberately constrained system of tarot suits of aesthetic principles omits.

Separate from the present discussion, however, the deliberately overlapping aesthetic tarot has distinct advantages as part of an artistic practice, and deserves further study from perspectives other than our present discussion of a general poetic of generativity.

5.1.4 Design Metaphors

In “Design Metaphors for Procedural Content Generation in Games” [192], Khaled et al. present a set of metaphors to describe generators. A generator’s role is compared with a human performing a similar role, so that an AI DESIGNER is metaphorically compared with a human designer. To distinguish between the machine and the metaphor, the machine’s role is designated in SMALLCAPS.

Part of their argument is that designers who don’t have experience with PCG have difficulty envisioning how to incorporate PCG into their design practice, so the metaphor is intended to “reduce the distance between design and engineering perspectives on PCG” [192]. Using familiar metaphors to bridge the gap is intended to help visualize the future of PCG research while also familiarizing the working designers with the state of PCG research.

One of the metaphors is the DOMAIN EXPERT, a system acting in a role similar to how a human domain expert is sometimes used in the production of serious games, to provide background expertise on a subject matter.

Going beyond Khaled et al.’s metaphor, we can extend this by contrasting it with the Artificial Intelligence concept of an expert system. An expert system emulates the knowledge of a human expert by encoding a decision-making process into a knowledge representation. Likewise, a procedural generator as EXPERT system is an expert on the thing it generates. That is, the generator is an expert on itself. Therefore, the definitive definition of what it means to be a dwarf in *Dwarf Fortress* [22] can be found in the data files and processes of the Dwarf Fortress source code. This line of thinking eventually led me to the concepts discussed in “Generators that Read” [200] (Chap. 6).

5.1.5 Generative Art and Effective Complexity

Games are far from the only place where generativity is used. In the fine art world, generative art has a long and venerable history. It predates computers, but has come into its own as computers have enabled new forms of generativity to be explored. One influential definition of generative art is that of Phillip Galanter:

> Generative art refers to any art practice in which the artist uses a system, such as a set of natural language rules, a computer program, a machine, or other procedural invention, that is set into motion with some degree of autonomy, thereby contributing to or resulting in a completed work of art. [113]

Galanter further applied the concept of effective complexity to generative art. Highly ordered systems (such as crystals or Penrose tiles) are simple. Highly disordered systems (e.g. randomization via cut-up techniques) are conventionally complex. But in information theory terms, effective complexity recognizes that highly disordered systems are nevertheless conceptually simple. Instead, Galanter traces a rough-peaked curve of effective complexity, placing evolutionary systems and artificial life at the peak of the curve (Fig. 5.1).

Galanter uses this to categorize generative art into highly ordered, complex, and highly disordered. Highly ordered generative art uses patterns and tiling, but little or no randomness. Disordered systems, in contrast, are highly disordered generative art: aleatory music, cut ups, etc. Standing in contrast to the relative straightforwardness of

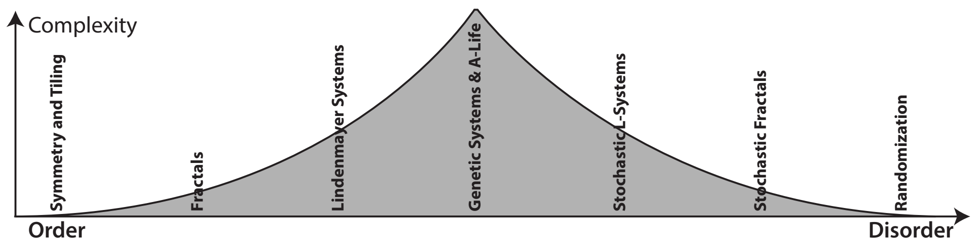


Figure 5.1: Galanter's continuum of generative art: complexity exists on a continuum from order to disorder. The peak of effective complexity is in the middle. [114]

tiled patterns and die rolls, Galanter's "complex systems" are self-organizing, emergent, and greater than the sum of their parts.

5.1.6 Procedural Aesthetics

Matt Stockham presents an aesthetic taxonomy in "Procedural Aesthetics" [338], focusing on identifying a toolkit of aesthetic devices, including defining regions in generated landscapes, generating rules of play, and using modular music.

In contrast, the aesthetic principles in the present chapter are not tied to any specific technique. There are many ways of expressing similar generator outcomes. To give one example: the choice of what kind of noise to use has an effect, but our focus is more on the effect that is achieved rather than on the specific algorithm used to achieve that effect.

However, studying the aesthetic effects of specific generative operations is a useful practical project for future work, which complements the model laid out in this chapter. Building a library of critique tied to specific examples will give us a better understanding of what has been achieved and what effects are possible. It deepens our ability to critique a watch when we have a better understanding of which gear the watchmaker should use.

5.2 Poetics of Procedural Generation

While there are many lenses that have been applied to generative systems, we still lack a vocabulary that describes most of the aesthetic qualities of generativity. While procedural generators share some of the same properties as other *expressive processes*, they also have aesthetic qualities that are specific to generativity. This chapter is not intended to present an exhaustive list—it is deliberately incomplete. It does present the aesthetic properties that I considered to be the most relevant for understanding the poetics of procedural generation.

It is also important to note that most of the properties presented here are not completely mutually exclusive: for example, gestalt section 5.5 and structure subsection 5.5.1 are in tension, but a generative system can exhibit both properties to a certain degree—and in many cases are more effective in combination than either would be on its own. Other properties can exist harmoniously.

Many generative systems use seemingly contradictory poetics in different parts of the same generative system, or have one generative system encapsulate another generative system that is based on a very different subset of poetics. Rather than a simple continuum, the poetics presented here can be thought of as a multidimensional space, with some of the dimensions being closely related while others are independent.

The properties discussed are divided into four lenses: Complexity, Form, Locus, and Variation. Each lens has several aesthetic properties that are in tension with each other, presented as two-dimensional lenses or three-dimensional triangular nomograms. [5] The diagrams are, of course, subjective—they are intended as rhetorical devices to provide context to the overall argument, rather than as a strictly objective metric of aesthetics.

Most of the examples in this chapter will be drawn from either videogames or from works made for the annual National Novel Generation Month (NaNoGenMo). Videogames are a broad category that exhibit a mass-media application of generative systems. In contrast, NaNoGenMo novels are participating in a specific artistic discourse, one which has been compared to the ideas in Ken Goldsmith's *Uncreative Writing* and the Dada art movement [163]. This gives us an opportunity to look at examples from usefully distinct perspectives.

5.3 Generative Aesthetic Complexity: Apollonian and Dionysian

Our first aesthetic lens, effective complexity, makes for an effective starting point for our examination of the aesthetic properties of generative poetics. The central tension in generativity is between randomness and order: generative systems need randomness for aleatoric novelty but without the ordered structure the novelty will lack

[5]Strictly speaking they are trilinear diagrams—though the strict definition has limited utility here: as subjective diagrams of aesthetic properties, deployed as visual rhetorical aids, they are not really something you should be doing rigorous math on anyway.

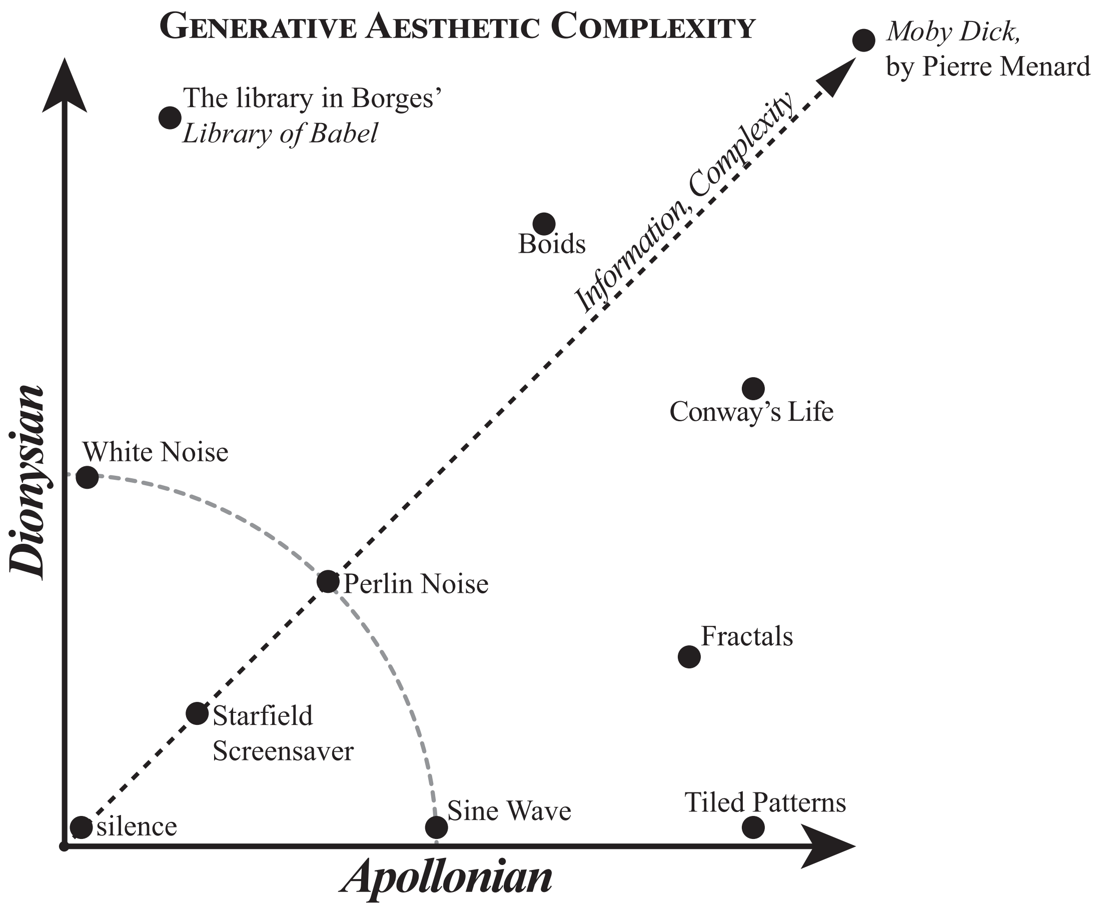


Figure 5.2: Diagram of Generative Aesthetic Complexity: rather than Galanter's linear continuum, the balance between the Apollonian and Dionysian aspects of generativity can be described as a two-dimensional space. Information and effective complexity increase towards the upper right corner, beyond which is the (hypothetical) computer-generated Great American Novel.

context and therefore will lack meaning. But the critical point is that, contrary to previous frameworks, it is not a one-dimensional spectrum. Order and chaos are *not* mutually exclusive.

To elaborate on the interaction between order and chaos, I introduced terminology derived from Nietzsche's aesthetic dichotomy of Apollonian and Dionysian impulses [257]. Unlike order and chaos, they are not conceived of as being mutually exclusive (Fig. 5.2). In their dialectic relationship, the reason and order of the Apollonian impulse is balanced against the emotion and chaos of the Dionysian impulse. Therefore, Apollonian and Dionysian elements exist *simultaneously* in generative systems. They are both in tension with each other and also support each other.

Marking our departure from Galanter's complex systems, I argue that the PCG system with the highest effective complexity is one that is both highly ordered and highly chaotic. That is, in my terms, systems have the most emergent complexity when they exhibit high levels of Apollonian generation, governed by structured rules, together with high levels of Dionysian generation, exhibiting noise and nondeterministic chaos.

There are, of course, also generative systems that exhibit low order/low noise. In the extreme corner we have non-generative systems that don't emit any artifact other than itself—in other words, pure inert data. Slightly more interesting are systems that, while not exhibiting high degrees of generativity, do feature a degree of structure or a degree of noise. A few use both: we can consider a simple screensaver, such as the

Microsoft Windows "Starfield" screensaver:[6] it has a structure (bright points always start near the center of the screen and fly outwards) and randomness (each point has a different position and direction). It is definitely generative, but does not have much effective complexity.

5.3.1 Dionysian Noise

Procedural generation in games often starts with randomness. While there are deterministic generative systems that construct an output via a parametric processes, in practice they often have their parameters controlled by some form of noise or randomness (such as a seed value). Clustered around the Dionysian pole, we find such things as aleatory music, the cut-up technique popularized by Burroughs (1961) [47], random tables in Dungeons & Dragons which are themselves descended from prior wargames [270, pp.311–314] and terrain generation with Perlin noise [267].

The most basic use of noise——the die roll, the coin flip——is uniform white noise, with an even distribution and each outcome independent from every other. But noise can also be expressive: both the distribution and the frequency can be adjusted to produce new effects. Adjusting the distribution is a very common method of altering the kind of randomness used——rolling multiple dice and summing the result approximates a bell curve, and countless 20th century wargame and roleplaying designs revolved around applying distortion curves to a die result in the form of a lookup table [270, pp.289–290].

[6]Sometimes titled "Starfield Simulation". A version of it was included in many editions of Windows (ending with Windows XP). It displayed small dots flying out from the center of the screen at random angles: `https://www.youtube.com/watch?v=r5TmP_tI5RI`

Less frequently discussed but equally practical is the spatial or temporal frequency of noise, also called the color of noise, in analogy to colors of light being determined by its wavelength [219]. The color of noise describes the timbre of the sound or the texture of its visual appearance. Red noise has less power in the higher frequencies, while blue noise is the reverse. Both appear in nature and have specific uses in audio production, visual effects, and procedural generation.

More coherent forms of noise, such as Perlin noise, have achieved such ubiquity in procedural content generation that they are sometimes synecdochically identified with procedural content generation itself.[7] Coherent noise is still noise, in that the output can be calculated independently for each point, but the mathematical structure of the noise generator means that neighboring points have values that are related to each other. Dice-metaphors are, of course, only one kind of randomness. A second frequently used paradigm is the shuffle, the metaphor of cards in a deck. This is still a random process, but with very different properties. In particular, sampling without replacement gives a designer much more control over the outcome of a generative system while still involving a large amount of hands-off randomness. Dionysian noise is more than mere chaos.

[7]Non-Perlin-noise generative systems are still effective. One example I like to use to demonstrate this is the Twitter bot Bot Ross, by R4_Unit [373], which tweets at `https://twitter.com/JoyOfBotRoss`. From the Bot Ross design notes:

> A few words about the philosophy. I really wanted to stick to my story about this being written by Bob in 1985. In particular, 1985 was the year Perlin fractal noise hit it big, so my first constraint was that I wanted the programmer of this to not know of Perlin noise. This is also a personal constraint since I think fractal noise is extremely over used in the procedural generation community which makes many projects look very much the same. In my day job, I am a probability theorist, and so I know the world of randomness is much larger than what is normally used. [373]

5.3.2 Apollonian Order

Of course, procedural generation is not just the Dionysian chaos of noise. What makes procedural generation distinct from just white noise is that it has coherence and structure that provides a logical relationship between the generation process and the result.

Generative art can exist with very little randomness, so long as the structure is interesting enough. The simple rules of Conway's Life [121] and the geometric tiling patterns in Islamic art, such as deployed in the game *Engare* [15], both follow fixed rules, but can create emergent results of startling complexity.

This complexity can also emerge in entirely deterministic systems. Chaos theory is the study of deterministic but unpredictable systems. Strange attractors [214, pp.130–141] and fractals [219] are created by ordered processes, but exhibit unpredictable emergent results.

This is complicated by the pseudo-random number generators (PRNG) that are used for most procedural content generation. A PRNG generates a number on request that is chaotic enough to be completely unpredictable to a human observer (or, for a crypographically-secure RNG, ideally to *any* observer). A truly random number can't be generated arithmetically [368]. Therefore, in procedural generation the creation of disorder is often dependent on order.

While some generative systems use other sources of randomness (most typically crowdsourced or via some kind of data source) the vast majority of generative systems in games use a form of order to create their randomness. The point is that none of these categorizations are pure. Generative systems can nest inside other generative systems or be assembled into generative pipelines that mix different forms of generativity.

5.3.3 Effective Complexity

Just as life needs both ordered growth and entropic decay, generativity needs both. All of the works that I will examine in this chapter will exhibit both, in varying proportions.

Even at the extreme edges we have things like completely enumerated spaces, as in Borges "Library of Babel" [34]: a place so disordered that it is ordered. And near the center we have Conway's Life giving birth to entire ecosystems while being entirely deterministic.

This tension is exhibited even in the construction of the generative systems themselves. As a practical matter, deterministic behavior in a generative system is usually desirable. Coordinating simulations between separate machines, reproducing the exact generation process, or having a predictable result all require that the same input results in the same output. Care is often taken to ensure that an entire generative pipeline is deterministic, without any true randomness. And yet even this ordered behavior is a fragile illusion: errors abound. A slight difference in timing might result in one choice being made differently, cascading true randomness through the generative system. Careful control of the generative process is required for this balancing act.

Therefore, the generativity is bound up in synthesis of Apollonian and Dionysian generativity, with the highest effective complexity to be found where they have achieved the ideal balance for that particular work.

5.4 Generative Form: Repetition, Gestalt, Individual

While the Apollonian and Dionysian aspects of generativity tend to be the most obvious aesthetic properties of a generative system, the less obvious properties can be equally important. For example, when we look at multiple artifacts that a generative system has produced, the relationship *across* those artifacts becomes meaningful. In what we term the *generative form* the generative system balances between the overall *gestalt* effect, the expressiveness of *individual* generated artifacts, and the effect of *repetition*.

5.4.1 Repetition

Repetition, in particular, has previously been under-discussed in comparison to its use in generativity. Emily Short's aesthetic tarot framework includes an articulation of the principle of *Mushroom*, which describes the repetition that grows on the forest floor of generativity:

> Mushroom is propagative and indifferent to the individual. As long as there are spores, the fungal principle is content. Mushroom-writing does not care about an individual instance of output and does not regret the loss of any element. Mushroom-writing thrives on decay, the breakdown of old structures, and the creation of new structures. Mushroom-writing is indifferent to consistency or to the profile of the resulting whole. It is unapologetic about repetitions. [311, p.89]

Here we can see that repetition can be a desirable aesthetic goal in itself—as well as the aesthetic effect of being obviously created by a generative process. Many existing generative systems make use of repetition to achieve this effect: Short considers Markov chains to be “mushroomy” [311, p.89].

In contrast to the naïve goal of having a generative system that is unable to be detected as a generative system, generative artists like Helena Sarin have leaned into work that is obviously generative.[8] Brian Eno’s dictum of “Whatever you now find weird, ugly, uncomfortable and nasty about a new medium will surely become its signature” [101, p.283] is particularly applicable here, in light of Eno’s history in developing generative music. The obviously generative nature of many procedural generators is what attracts our interest. And the unique thing that a generative system has that no human can ever replicate is that the artifacts it produces *were made by a generative system.* Leaning in to the machine-made nature highlights the Mushroom aesthetic.[9]

Repetition as an aesthetic device is a well-known element in artistic composition. Many periods of architecture, ranging from Classical to Gothic to Modern, involve extensive use of repeated forms. Classical architecture used repeating columns: Vitruvius prescribes using the column itself as the indexed measurement for the module [366, 4.3.3, 1.2.2], with a particular focus on proportional relationships. There are many different ways that repetition can be deployed, using the texture of the repetition for aesthetic effect.

Many poetic forms use repetition extensively, ranging from parallel imagery

[8]For example, Sarin’s *The book of GANgesis* [120, 302]

[9]This is also related to a very interesting conversation I had with Kate Compton about liquid versus solid generation, of which this margin is too small to contain a complete explanation [60]

to repeating words verbatim, to the point that some poetic structures, such as villanelles [109,110],[10] are premised on exactly repeated phrases. Some generative systems use this principle directly: the popular @infinite_scream Twitter bot [289] tweets variations on "AAAAAAHHH" for followers to interact with.

Unlike modernist or classical repetition, generative repetition can be imperfect—in fact, the existence of the repetition makes the imperfections and asymmetries more significant.

Generativity allows us to recapture the form of handmade repetition. In contrast with the identical mass-produced objects of modernity, generativity can approximate the subtle variations of the handmade artisanal craft production. The Arts & Crafts movement valued this aesthetic property highly. John Ruskin viewed the inclusion of imperfection as a moral good:[11]

> The second reason is, that imperfection is in some sort essential to all that we know of life. It is the sign of life in a mortal body, that is to say, of a state of progress and change. Nothing that lives is, or can be, rigidly perfect; part of it is decaying, part nascent. The foxglove blossom,——a third part bud, a third part past, a third part in full bloom,——is a type of the life of this world. And in all things that live there are certain, irregularities and deficiencies which are not only signs of life, but sources of beauty. No human face is exactly the same in its lines on each side, no leaf perfect in its lobes, no branch in its symmetry. All admit irregularity as they imply change; and to banish imperfection is to destroy expression, to check exertion, to paralyse vitality. All things are literally better, lovelier, and more beloved for the imperfections which have been divinely appointed, that the law of human life may be Effort, and the law of human judgment, Mercy. [297, p.32]

[10]On a slight tangent: I found Amanda French's history of the villanelle interesting as both a re-examination of the history of the form and as a way to present academic research online: `https://villanelle.amandafrench.net/`

[11]Admittedly, Ruskin would probably not appreciate machines taking on this sense of imperfection, though I would argue that the way a generative system designer's intent is expressive in the artifacts created *does* fulfill the welcoming of variation in the Gothic aesthetic. In particular, the irregularities can imply an intent (Sec. 5.5.1.1). I still doubt Ruskin would agree.

Individual generated artifacts can be made unique, even if they are not novel in the aggregate: bolts on an airplane wing generated with minute variations, as in Denis Kozlov's "Project Aero" [195]; a unique color scheme, as exhibited by the creatures in *Spore* [225]; or the color of a star in *No Man's Sky* [153]. It returns a sense of uniqueness to the individual manufactured object, though this time it is the individuality of the machine rather than evidence of the direct human hand.

5.4.2 Gestalt

Extending this view, generativity can create a fractal repetition, a repetition where no individual snowflakes are the same, but the combination creates a formal gestalt impression, a generative snowbank.

This *gestalt* effect, where individual points are less important than the effect of the whole, is an underappreciated aesthetic outcome of generativity. This is one of the ways in which procedural generation can be generative in another sense: having the qualities of plant growth. The aesthetics of generativity are often organic, resembling the verdant emergent forest undergrowth.

A procedurally-generated forest is a good example of the gestalt aesthetic: the exact placement of individual trees is not important as long as there are enough plants to convey the idea of a forest. The gestalt is what matters, not any specific individual or the exact shape of the overall forest.

For the gestalt, both the individual tile and the overall pattern of tiles on an infinite tiled floor are a smaller part of the aesthetic experience. Instead, the viewer's aesthetic experience of rules that create the pattern is most strongly felt through the perception of something else entirely—a Platonic form that can only be glimpsed indirectly in this liminal space that is neither the whole nor the individual part.

The designers of *No Man's Sky* [153] originally intended for players to approach its planets as gestalt experiences. At launch, each planet had a single environmental biome and a roughly-even distribution of features specific to that planet. The idea was that this would let players quickly recognize what kind of planet they had found, encouraging exploration of new planets [115, 22:14]. The player's loss-of-interest pattern-recognition takes hold somewhere between visiting a single location (often too brief to grasp the entire pattern of the planet) and exploring the entire planet (virtually impossible).

Similar to the Apollonian/Dionysian dialectic, repetition is the Apollonian mirror of the more Dionysian gestalt-perception: the chaos of the forest-gestalt compared to the ordered rows of an orchard. In both cases, the *overall perception across many artifacts* dominates the aesthetic rather than any single artifact.[12]

[12]This is complicated by generative systems that are nested within generative systems, which in practice is most generative systems: Is the artifact of a forest generative system the placement of each tree, or the forest as a whole? It depends on the particular generative system, particularly because some generative systems can create things that they have no *explicit* representation of in their internal model but do have an *implicit* representation. This is one of the rare ways in which the fussily-precise computer can represent a fuzzy ontological concept. There might not be any forest data structure or function call, but the generative possibility space of the generative system contains a region that makes what we would recognize to be a forest.

5.4.3 Trees, Forests, Orchards

In summary, the *generative form* of a generative system's output can be described in terms of three axes (Fig. 5.4).

First, the importance of the *individual* artifacts it generates, such as a tree generator that produces many perceptually unique trees. This is, in some ways, the naïvely traditional procgen aesthetic: attempt to make each generated artifact unique enough to appear not to be a computer-generated artifact. However, the individual artifact is, often, quite important, and some generative systems, such as the galaxy generator in *Elite* [38] or the map generator in *Exile* [168] use a fixed seed to ensure that they only ever produce a single unchanging and all-encompassing artifact.

Second, the overall *gestalt* effect, as in a forest generator. Rather than producing a single artifact, it produces many artifacts and uses the relationship between the objects to convey meaning. The forest generation in *Age of Empires II* [102] stamps down tree sprites from its small, fixed collection of tree sprites (Fig. 5.3). Once placed, the trees don't know about the other trees, but the player (and the AI) still reads it as a forest. This also applies to more abstract *forests*: any generative system that creates a collection of things and then derives its aesthetic effect from the overall gestalt can be thought of mnemonically as a forest generator.

Third, in contrast to both individual aesthetics and gestalt effects, a generative system's aesthetic effect may stem from its use of repetition. An *orchard* of identical trees uses the *lack* of variation to create an order that contrasts sharply against the

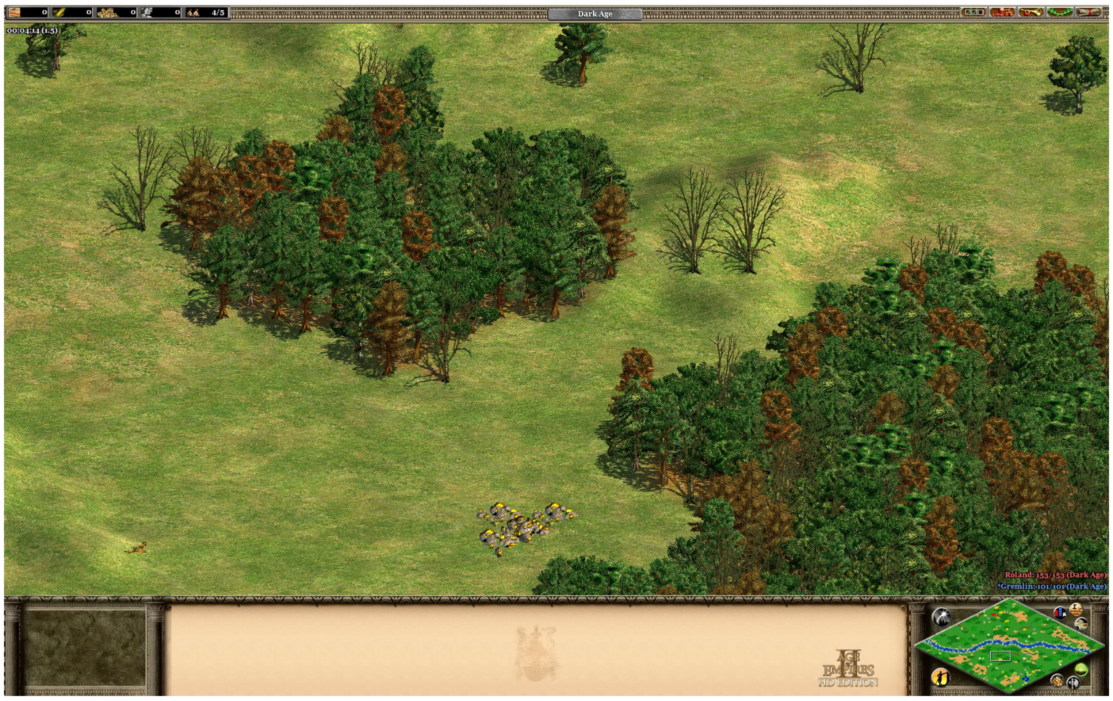


Figure 5.3: Trees in Age of Empires II: the original release of the game represented trees by means of a small number of tree sprites. The forests are procedurally generated, but the individual trees are fixed images. (Although the distinction isn't visible on this game map, this screenshot is from the HD edition [103], which includes a greater variety of tree sprites from the expansion packs for use across different themes or biomes but still uses the same basic approach within a given biome.)

noise of the forest. This effect creates the feeling of a cause-and-effect relationship, even if the actual cause is quite different than the felt cause.

The three axes of *individual* trees, *gestalt* forests, and *repetition* in orchards collectively make up what I've labeled the *generative form*.

5.5 Locus: Structure, Surface, Gestalt

Rotating our perspective, we can contrast the form of the generative system against the generative system's aesthetic *locus*:[13]

Where Form is about the space of artifacts, Locus is about the system that generates those artifacts. Specifically, where the system's focus is located, and how an outside observer perceives that system. The focus can be on the surface (the experience of artifacts); the structure (the indirect perception of the system's inner workings); or the the gestalt (the ideas perceived in the relation between generated artifacts). Form and Locus are parallel but not identical. They both share the gestalt aesthetic (a generative system that has gestalt form frequently has gestalt locus, to the point of deliberately sharing a name). Surface is often (but not always) associated expressed through individual artifacts, and an orchard makes structure very visible. But you can equally express structure through a handful of individual artifacts (e.g. engravings in *Dwarf Fortress*).

The generative gestalt aesthetic is an abnegation of the immediate details of

[13]This idea of a generative system's "locus" is coined here in a very loose metaphor with Rotter's "locus of control" [296, p.489-493]

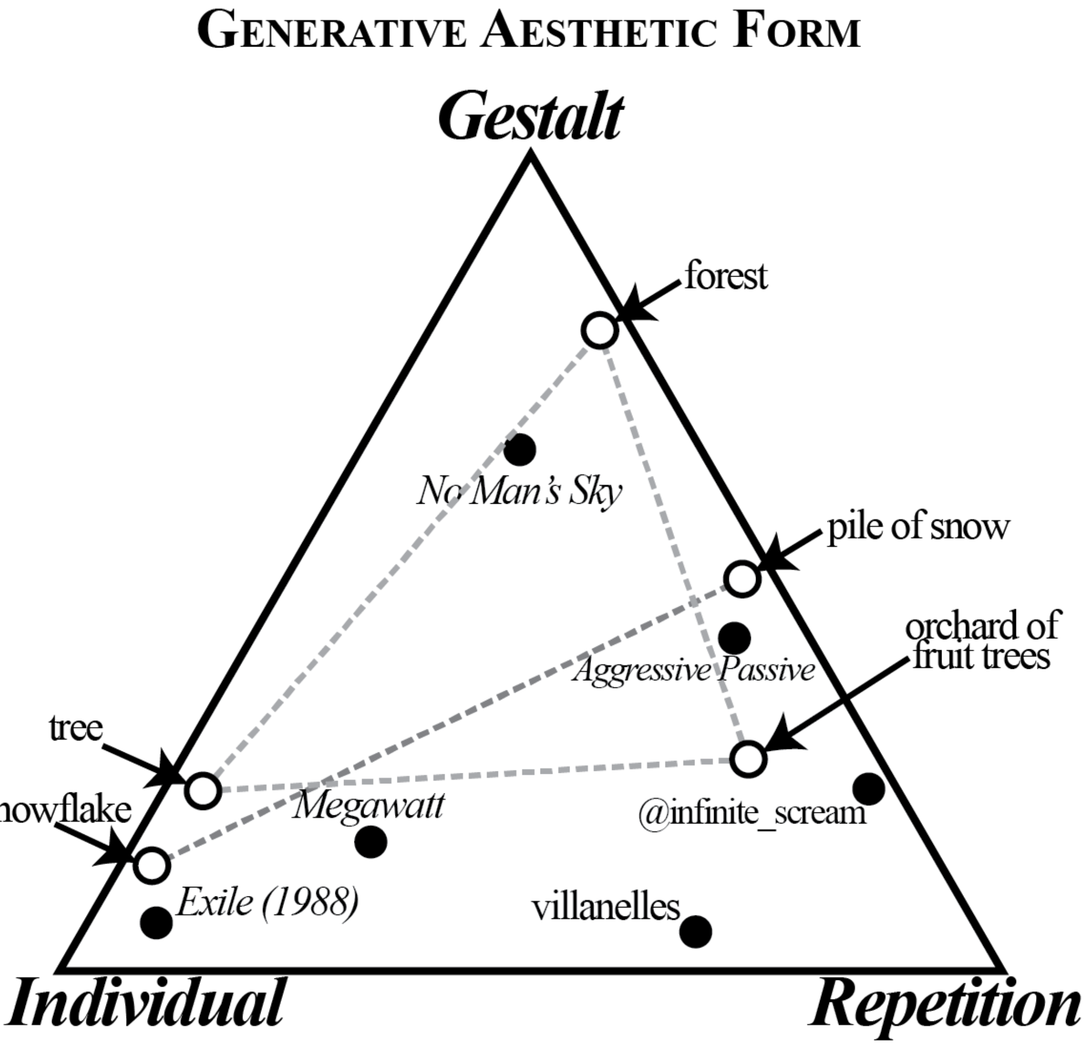


Figure 5.4: Generative aesthetic form: the tension between repetition, individual artifacts, and the generation as a whole. A given generator can emphasize individual artifacts (such as making individual trees with as much variety as possible), or can focus on the gestalt effect that many artifacts have when clustered together (a forest of trees, with each individual tree not being as important as the overall effect), or it can make use of the contrast between different artifacts (putting many similar trees together in a regimented orchard emphasizes both their similarities and their differences and makes the aesthetic focus about the relationship between the artifacts, instead of the individual tree or the group as a whole).

the sensory experience in favor of the hidden effects emanating from that experience. But there are also generative systems which are concerned with the macro-structure of the system itself. For Short, this is partially subsumed under the principle of Salt: "For the principle of salt, the machine-that-writes matters more than the thing-written" [311, p.88]

While often associated with Apollonian order, a Dionysian generative system can also be structurally expressive: in Jonathan Basile's digital implementation of "The Library of Babel" [21] the sheer scale of the permutations transcends the noisy gestalt and turns the reader's focus to the ideas of infinity embedded in the architecture of the generative system.[14]

Short additionally associates Salt with the grammars she created that expressed specific ideas like kinds of cheese or scents of perfume [311, p.88]. This echoes the design metaphor concept of an EXPERT SYSTEM: the system is an expert about itself. The map is literally the territory.[15]

Generative art that borders on conceptual art tends to be about expressing ideas through the structure of the generative system's processes. To illustrate this, let us consider three NaNoGenMo novels: Aaron Reed's *Aggressive Passive* [287], Nick Montfort's *Megawatt* [239] (2014), and Andrew Plotkin's *Redwreath and Goldstar Have Traveled to Deathsgate* [273]. One of the (only) two NaNoGenMo rules is that code must be made public. This has led to the code itself being part of the intended message, either

[14]Note that *gestalt* is used for both Locus and Form: the use of the same name is deliberate, because in my view these closely-related aesthetic effects are the same thing.

[15]On this point, see also Martin O'Leary's NaNoGenMo novel *The Deserts of the West* [261] which itself references Borges' "On Exactitude in Science" [33].

in a literate programming sense of the actual source code being intended to be read, or (more conceptually esoteric but practically approachable) as the way their processes operate.

NaNoGenMo generators often are capable of outputting multiple novels. A few, as in the case of *Megawatt*, produce a deterministic output that nevertheless foregrounds the process of generation. *Megawatt* takes a Samuel Beckett novel and extrapolates it:

> In the new novel, rather, they are intensified by generating, using the same methods that Beckett used, significantly more text than is found in the already excessive *Watt*. [239]

Redwreath and Goldstar operates by expanding a grammar recursively to create a conversation that consists of characters answering questions with more questions, in a parody of both Steven Brust and Alexandre Dumas. *Aggressive Passive* uses a similar structure to tell stories about housemates blaming each other for not doing chores. In both cases, the individual lines don't particularly matter, and instead the meaning is invested in the structure of the generative system. The process of the system as a whole is the message, with the artifact as an almost incidental realization of the concept.[16]

5.5.1 The poetics of structure

This emphasis on the structure of the generative system and its processes is a lens that lets us understand a common element of play in games that feature generative systems: the player can infer how the hidden generative system works and

[16]Which is an apt demonstration of generative art's relationship with conceptual art: as I've occasionally said in conversation, "Generative art is conceptual art except someone went and actually made the damn thing."

draw conclusions from that. In my experience, designing with this in mind is one of the signs of a maturing generative designer.

Many generative systems have deliberate tells, such as the way that dungeon generation in NetHack [255] makes it easier to predict that a secret door might exist. Understanding how the level generator lays out space makes this prediction possible [49]. Knowing that there should be a room in the space where nothing is visible creates an aporia that facilitates agency [179] without ever explicitly instructing the player where to search for secret doors in the level's negative space.

There are many ways to achieve this. One way is to have a predictable distribution, such as the way weapon and item generation in *PLAYERUNKNOWN'S BATTLEGROUNDS* [278] can be predicted from the kind of building they are found in; or the patterns that raw diamonds follow in *Minecraft* [238], only appearing in clusters at defined depths [235, 236]. Another way is to enforce consistent generative rules, such as the way that most levels in *Spelunky* [213] always generate the exit lower than its entrance [385, p.34–36]. Yet another way is the inclusion of guaranteed or fixed content in an otherwise dynamic generator, such as the way that *NetHack* [255] has a number of special-case levels at predictable but slightly varying depths [49, 256].

Aesthetic properties, such as symmetry [132, 3.1], can enable the player to infer information before actually encountering it. This is an effect that can be carried over from wayfinding in real-world buildings, where properties of correspondence, compatibility, completeness, and so forth are considered when analyzing building design [51]. This speeds up the process of the player familiarizing themselves with the system, and allows the designer to surprise the player by breaking the pattern.

5.5.1.1 Implied Cause and Effect

One common theme is that exhibiting a strong aesthetic property implies a cause-and-effect relationship to the viewer, regardless of whether it actually exists. Similar to a visual effects artist adding a shiny edge to an object to imply a history of constant wear, repeating a motif or having cohesive content implies a history.

This can be stronger, of course, if the implied cause is something the generative system teleologically implements: a temperature simulation affecting biome distribution will mean that the secondary implications will propagate indirectly (e.g. rain shadows will naturally appear when combined with wind direction and altitude). But they can also be achieved through a non-realistic teleological process: using the same source of Perlin noise for different effects gives them a correlation even though that correlation doesn't actually match the implied cause. (See also the discussion of teleological generative systems in Sec. 6.2.2).

More broadly, a generative system that produces artifacts that exhibit regular features like symmetry implies intent in the design. In a sense there *is* intent: by making a generator that can care about symmetry (either explicitly or emerging from the generative possibility space) there is an actual design intent, even if it isn't quite the exact kind of intent that is being implied by the visible aesthetics.

5.5.2 Hierarchy and Distribution

Hierarchy is an important aesthetic principle in many disciplines, including visual hierarchy in graphic design [54, p.48] and in architecture [7, Sec. 114]. One way that hierarchy is expressed that is more unique to procedural generation is via the distribution of the generated artifacts.

Both frequency and rarity have specific effects on the player's perception. Rare artifacts, such as the villages and strongholds in *Minecraft* [238], are perceived as more important and examined individually. They have an expectation of uniqueness, although repetition at long distances can also engender a sense of ritual. In contrast, frequent results are generally closer to a gestalt effect, forming the background as a contrast to the figure of the rarer results.

Distributions that follow non-linear or biased curves are often more interesting than linear probabilities: pushing the generative system to extremes gives more interesting results. Just as animators use easing to create more pleasing motion, rolling a pair of dice and adding them together creates more interesting random results than rolling a single 11-sided die would. Making the highs higher but rarer creates contrast that makes them stand out, with very unusual results acting as landmarks.

Adjusting the weighting or distribution of outcomes is often an important part of the design process when working with procedural generation. In *Voyageur* [87] (Dias, 2017a), Bruno Dias used both a salience system and hand-tuned weightings to control the frequency at which pieces of content would be used in the generator [86].

This also highlights how the aesthetic effect of a generative system is often indirect: just as game players talk about the "feel" of playing a game [348], when we interact with a generative system enough we can develop a proprioception for how it will respond. The invisible parts of the generative system can become part of our sensory experience.

5.5.3 Individual: The Artifacts as Surface

Of course, individual artifacts of the generative system can be important in themselves, apart from how they relate to other generated artifacts. After all, the artifacts are usually the only surface through which the player can observe and interact with the generative system.

This borrows from Noah Wardrip-Fruin's use: "the surface of a work of digital media is what the audience experiences: the output of the processes operating on the data, in the context of the physical hardware and setting, through which any audience interaction takes place" [370, p.10]. I extend this to the idea that individual sub-processes have their own virtual surfaces: the map generator in Minecraft [238] can only be experienced through the intermediate surface of the voxel blocks.

Therefore, the individual generated artifact is in triangular tension with both the form and the locus of the system. Depending on the generative system, individual artifacts can be anywhere from nearly anonymous (a tree in a forest) to the only visible artifact: the 1988 BBC Micro game *Exile* [168] uses a fixed-seed generator to create its unchanging map, enabling it to fit a sprawling cave system on-disk [203, p.92].

As with the Dionysian and Apollonian poles being distinct from complex systems, a generative system that focuses on individual artifacts uses both gestalt and structure, but is ultimately something else.

In Short's terminology, "Egg represents the egotistical, the view of the self as unique and special" [311, p.87] and "The egg principle is the principle of consensus, the principle of combination, or the principle of the authorial self" [311, p.96]. Notably, Short also uses this to include human curation and mixed-initiative generation.

In a generative system that emphasizes the individual artifact, the player can be expected to scrutinize each individual result more closely. An individual-focused generative system puts more depth into the generation of each artifact. Here, quality is far more interesting than quantity. Individual artifacts can, nevertheless, participate in larger systems. While the artifact as a whole might be unique, parts of it can reflect the system that made it. The engravings in *Dwarf Fortress* [22] are each individual and unique (when created by a skilled dwarf), but the images they depict are linked to the history of the world and the events in the fortress where they are engraved [94]. The engravings are individual, and that individuality puts them in a cohesive context (Sec. 5.6.3).

5.5.3.1 Rarity and Disposability

One common aesthetic problem in generative art, and generativity more broadly, is how you make the viewer care about *this particular artifact*. The machine can produce an infinite number of disposable artifacts, and as Borges aptly describes in "The

Library of Babel" [34] fatigue rapidly sets in. One move is to avoid the question entirely, retreating into the gestalt aesthetic. However, if we do want to make someone care there are a few approaches:

The most obvious is artificial scarcity: intentionally limit how many of the artifacts the user can experience. Even a relatively gentle limit can invoke this feeling: there is a qualitative difference between prompting a language model on demand versus having to wait for a twitter bot to periodically post.

A second way is to give the artifact context: this isn't just any old artifact, this is *my* artifact. Or, it is related to the things around it (physically or metaphorically) making it the artifact in *this* place and not *that* place.

A third way is to create a ritual around interacting with the generative system, which combines the other two ways: by making a player enact some form of ritual around the result, it creates both context (my result) and scarcity (because I only did this one). Decision-making through coin-flips, tarot reading, rolling dice in a roleplaying game: in each case there's nothing physically stopping you from trying again for a different result, but the ritual has imbued that *particular* first result with meaning.

It is worth studying sports on this point: they are rife with engines for getting viewers to attach importance to random things, all the way from big things like arbitrary team associations right down to particular transient events. Which is likely one reason behind the success of *Blaseball* [16], as it is particularly good at turning sports rituals up to 11.

Doubtless there are other approaches; ritual in particular is a fruitful area for generative artists to explore.

5.5.4 Generative Aesthetic Locus

In general terms, gestalt aesthetics tend to dominate in pure Apollonian or pure Dionysian systems, such as tile patterns or white noise. Structured aesthetics, such as *Spelunky's* level generator, tend to be unbalanced mixtures of the two. Complex systems, such as the fixed-seed map generator in *Exile*, tend to favor individual aesthetics, because a single artifact has more scope to display internal variation. But these aesthetics are ultimately orthogonal to the Apollonian/Dionysian tension and can be found in many different permutations.

In summary, the locus of the generative system can be described as the balance between the surface of the individual artifacts it generates, the structure of the process that it uses, and the ideas that the player perceives in the gestalt of the things generated (Fig. 5.5).

We can also define these three aesthetic properties by what they are *not*: a surface aesthetic is less conceptual or abstract; a gestalt aesthetic is less actionable or functional; and a structure aesthetic is less sensory or immediately emotional.

5.6 Variation: Multiplicity, Style, Cohesion

A third lens involves *variation*. A common reason to use a generative system, after all, is if you want a lot of different things. But there are different kinds of variation, and three ways we can partition the aesthetics of variation are multiplicity, style, and

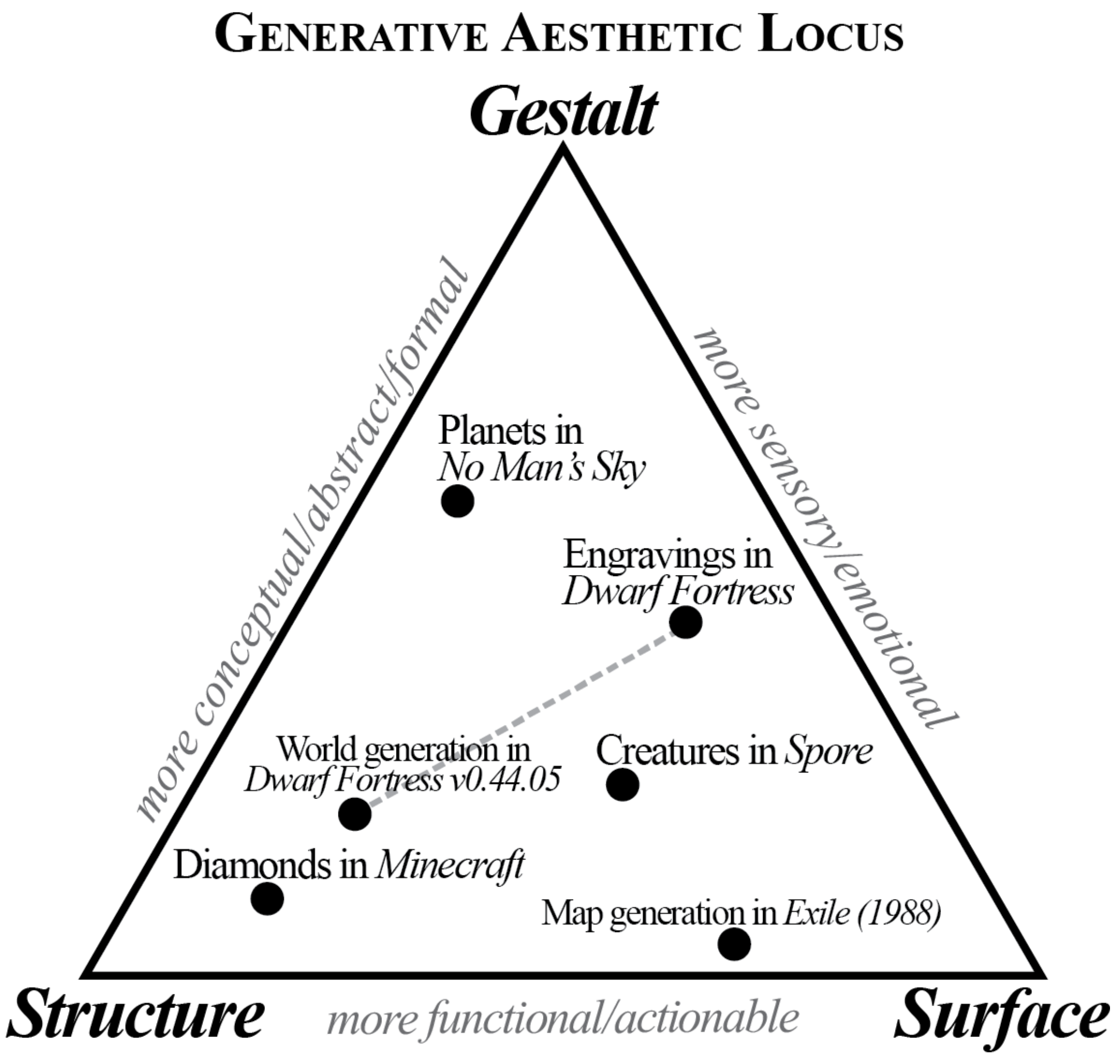

GENERATIVE AESTHETIC LOCUS
Gestalt
more conceptual/abstract/formal
more sensory/emotional
Planets in *No Man's Sky*
Engravings in *Dwarf Fortress*
World generation in *Dwarf Fortress v0.44.05*
Creatures in *Spore*
Diamonds in *Minecraft*
Map generation in *Exile (1988)*
Structure
more functional/actionable
Surface

Figure 5.5: Generative aesthetic locus: the player's experience of a generator is a balance between the *structure* of the process, the ideas and associations the player perceives indirectly in the *gestalt*, and the immediate *surface* interface of individual artifacts that the player can interact with or directly observe. The exact location of the examples in the diagram are arbitrary but illustrative: *Exile*'s map generation is mostly surface, since it produces exactly one deterministic artifact, but it nevertheless encodes certain relationships in the structure of the generator that are reflected in the details of the cave system that it generates.

cohesion (Fig. 5.6).

5.6.1 Multiplicity

The most obvious form of variation is a generative system that can produce a wide variety of very different results. The naïve form of this is popular: for example, players and press misunderstanding what it means to say that a generator has 18 quintillion outputs [156]–which usually only means that it is using a 64-bit seed for its random number generator. The seed is the upper bound; the actual amount of variation is limited to the generative possibility space, which is usually quite a bit smaller. Just as the effective complexity of white noise is lower than its incompressibility implies, the perceptual uniqueness of a generative system can be much lower than its theoretical variance.

Kate Compton has framed this as the "10,000" bowls of oatmeal problem [57]: an oatmeal generator can produce an incomprehensibly vast number of different ways to arrange the individual oats in a bowl, but they all get subsumed into our gestalt impression of indistinguishable bowls of mush.

Expressive range analysis [308] is, in part, an effort to measure the *effective variation.* If we can select the right metrics, we can use it to measure the effective variation across the generator's generative possibility space.

To maximize the effectiveness of multiplicity, the variation should be closely tied to both the visual presentation and to the other systems in the game (or, more broadly, to the viewer's way of perceiving the artifacts and the context that the gener-

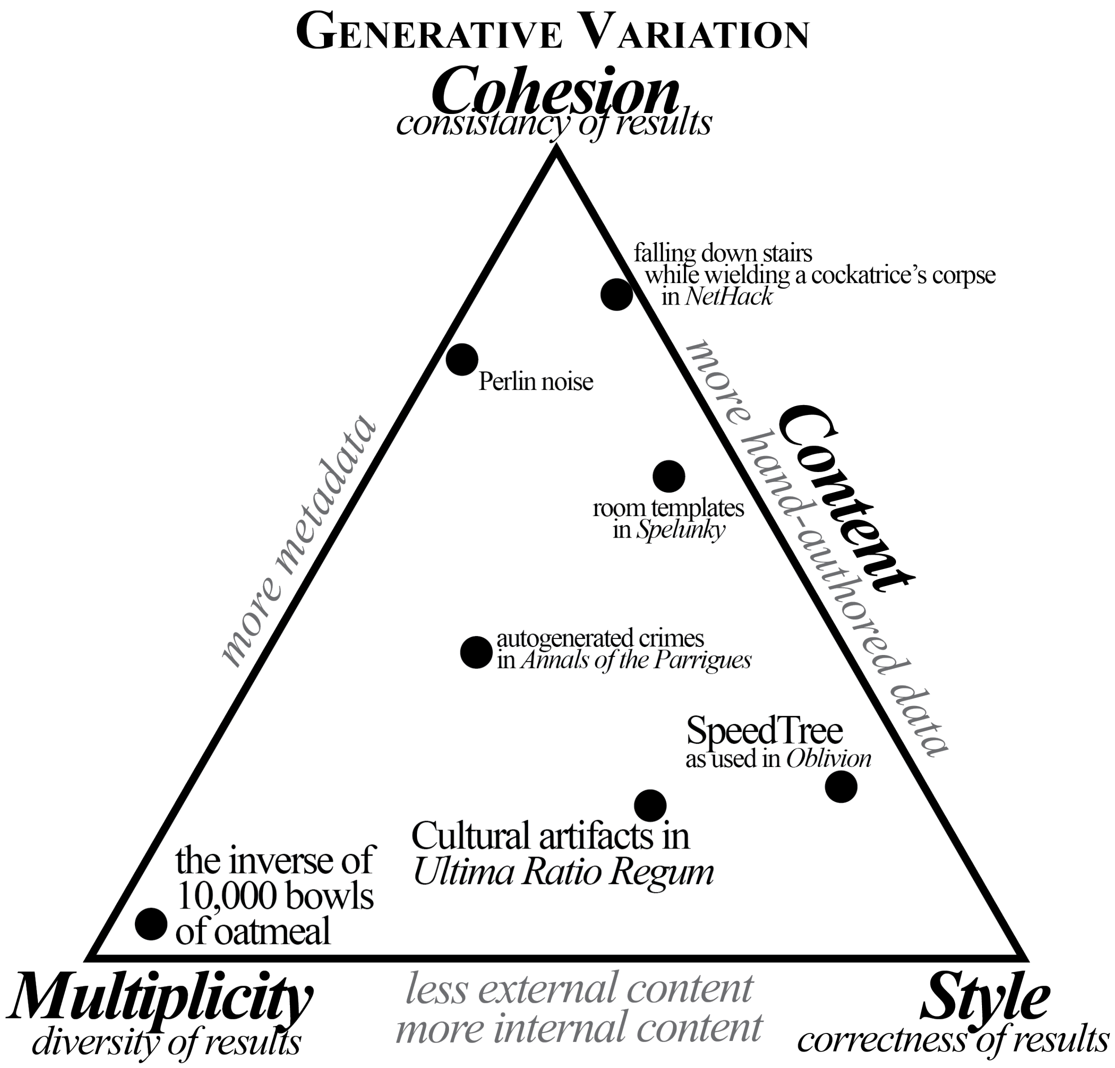

GENERATIVE VARIATION
Cohesion
consistancy of results
falling down stairs while wielding a cockatrice's corpse in NetHack
Perlin noise
more metadata
more hand-authored data
Content
room templates in Spelunky
autogenerated crimes in Annals of the Parrigues
SpeedTree as used in Oblivion
Cultural artifacts in Ultima Ratio Regum
the inverse of 10,000 bowls of oatmeal
Multiplicity
diversity of results
less external content
more internal content
Style
correctness of results

Figure 5.6: Generative aesthetic variation: Variation in a generator can exhibit multiplicity, style, or cohesion. Multiplicity is about how diverse the results are. Style is about how correct those results are for the criteria beyond the immediate parameters being fed into the generator, and Cohesion is about how different artifacts from the same generator fit together. The italics on the edges describe general tendencies I have observed in those corners: A cohesive-emphasis generator tends to use more metadata and more hand-authored content, a style-emphasis generator tends to rely less on external content sources and more on encoding the knowledge of the artifacts being generated in the internal structure of the generator (while still tending to use more hand-authored data than a multiplicity-emphasis generator).

ative system exists within). Effective variation matches the importance of the variation with the significance of its presentation.

For example, human visual perception strongly relies on contour. When subjected to a blank, diffuse field of vision, such as in a homogeneous Ganzfeld, the observer usually ceases to perceive color [55]. Artists and animators use this in character design: silhouette is a stronger signal than interior color. We can likewise apply it to the design priorities in generative systems. At launch *Elite: Dangerous* [112] featured a space station generator that was capable of creating a wide variety of different configurations for their large, spun-axis stations. However, due to the way that the docking mechanics worked, players typically observed the station from end-on. Since the station designs were roughly cylinders, and the player was focused on the docking port in the middle, from the players' perspective the space stations had the same silhouette. This led to the wide range of variation being much less noticeable in practice [180] (Fig. 5.7).

5.6.2 Style

Distinct from multiplicity's concern with the number of different variations across many artifacts, variation also exists from the individual artifact's perspective. In other words, this particular artifact could have been different, so why is it what it is?

This variation can be concerned with constraining the artifact to conform to a set of constraints, or it can enable the artist to shape the artifact directly. The latter is variation that gives an artist control of the parameters of the generative system, selecting one specific tree out of the generative system's possibility space. An example

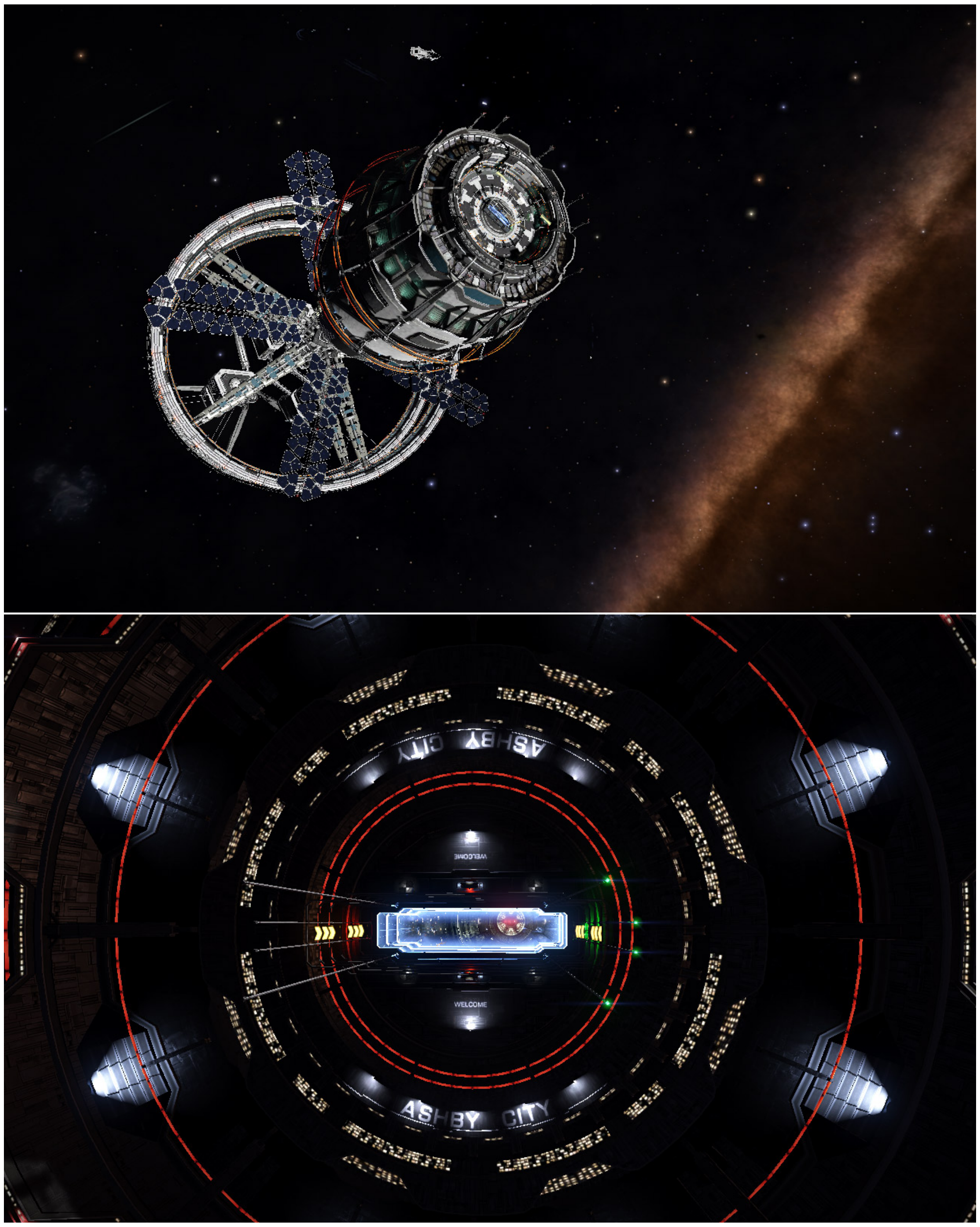


Figure 5.7: *Elite: Dangerous* [112] has a station generator that can create a vast array of different designs for the stations, including different configurations of rings, solar arrays, and other elements (above). However, at launch, every station had an identical docking port and silhouette (below), and since the vast majority of the players' interaction with the station exterior involves carefully maneuvering through the docking port, the perceived variation was relatively low, making its variation have low *multiplicity*.

of the former is a tree generator that takes the surrounding space into account and adjusts the tree accordingly.

An instance of both in common practice is the widely used SpeedTree middleware. Deployed in the production of games like *Elder Scrolls IV: Oblivion* [26], artists use SpeedTree's generative system to rapidly generate many trees, which the artist can then adjust manually as needed, enabling them to rapidly place trees and vegetation [82]. SpeedTree can also use geometry forces to have the tree interact with the space it is growing in [166].

We can term this aesthetic property the *style* of the artifact. We define a generative system's style property as the degree to which the generative system is able to adjust an artifact to conform to an objective.[17] In contrast to multiplicity's measure of the diversity of artifacts that can be output, style is about getting the *right* generated result.

Dwarf Fortress [22] uses this frequently, generating poetic forms, books, legendary artifacts, and the engravings mentioned above. Using content from the simulated world history, they exhibit style that reflects both that history and the individual history of the fortress the player has created. Similarly, *Ultima Ratio Regum*, Mark R. Johnson's in-development roguelike, focuses on *style* as a way of impressing meaning into the culturally-influenced artifacts and the AI of NPCs [172].

[17]This use of "style" is inspired by Stella Mazeika's work on style generation [226].

5.6.2.1 Effective Variation

Like effective complexity, effective variation is largely concerned with finding a balance between multiplicity and style (while also, as we'll see below, keeping the results coherent and consistent).

As mentioned above, one way to measure the variation of a generative system is through the process of expressive range analysis, via a metric that is an emergent result of the generative system rather than one of the parameters to the generative system [326, p.1]. Exactly which emergent result to measure depends on the designer's goals for the generator and the structure of the generative system itself.

Emily Short uses "Venom" to name a concept related to effective variation:

> Venom is meant in the sense of toxin, hallucinogen, bitterness, acid, etching, numbness, drugs, and release from the mortal coil. Venom represents that which is destructive, fictive, cruel, lovely, playful, unreliable. Poisonous things come in jewel colors. The principle of venom permits the use of connotation rather than denotation. [311, p.93]

When "writing venomously" Short recommends that, when adding variation to text, the focus should be on finding the most statistically implausible, meaning-bearing words in a sentence. When writing a template describing how a crime was committed, "the Principle of Venom suggests the use of a large, autogenerated corpus to supply the crime, rather than relying on the author's own imagination" to produce results that "are genuinely outside the expectations of the author" [311, p.95].

Separately, Short suggests that the number of conceptually-surprising variations should be kept to a few effective ones, to avoid creating overly-complex clashing images. Over-variation is prone to a kind of metaphor fatigue.

Elsewhere, Short suggests that one way to avoid the oatmeal bowl problem is to tie the generated results back into the other systems in the game. This can either be in a structured way—which she compares to the combinatorial effects of cards in a deck-building game—or in a low-level gestalt approach, with small-scale but persistent effects [312].

5.6.3 Cohesion

Cohesion is about how different artifacts from the same generative system fit together. The perception of cohesion is dependant on the consistency of the results: does it look like these artifacts belong together?

A simple form of cohesion is exhibited by Perlin noise: the value at one point in the plane is similar to neighboring points. Unlike incoherent white noise, there values appear to relate to each other. More complex cohesion requires the appearance that different part of the generated artifacts support each other, or at the very least, don't contradict each other. This can be through a deep simulation—*Nethack* [255] ensures cohesion by detailed implementation of cause-and-effect relationships—or through embedding the cohesion information in the topology of the system. *Spelunky* [213] ensures that its room templates fit together by making its generator correct-by-construction [385, pp. 34–40].

The correct-by-construction path from entrance to exit in a *Spelunky* level is a good example of how information can be implicitly embedded in a generative system. The generative system only places a room on the path if the player can traverse it, but it doesn't need to calculate this: the information was decided by the initial level generation, so no additional pathfinding is needed to make a cohesive level, so long as the room templates conform to their intended shapes.

5.6.3.1 Grounded Variation

One misconception some players and developers have is that the ideal procedural content generator creates everything from scratch, with no external content. This is a category error: every generative system uses external content. In some cases that input can be supplied algorithmically via another generative system, but every generative system needs to be grounded in designer-created input (cf. Sec. 8.1.2). While it is possible to replace all textures with mathematical functions, the design of the texture-function itself is a form of expert content authoring.

Hand-authoring multiple different generators can sometimes be the most effective way to create a generator with wide variation. The easiest way to increase a generator's expressive range is to make a second generator that functions differently. Likewise, hand-authoring content is often the most efficient way to create certain kinds of inputs—many text generators exploit their specific authoring context to constrain their text to a cohesive whole.

5.6.3.2 Data: Content

All procedural generators use some form of data. It can be via algorithms, parameters to generative functions, or some more complex hand-authored content. Moreover, the data the designer chooses to include is intrinsically bound to the effect of the generator. In terms of the Generative Methods model, generators have *inputs* (Sec. 3.6).

While it is useful to consider the system that transforms the data separately from the corpus of data it uses as input, the data itself often has a structure that affects the distribution of artifacts in the generative possibility space.

Take a Markov chain text generator as an example: the algorithm remains fixed but the distribution of words in the corpus determines the distribution of words in the output. A change in the corpus has significant effect on the generated output. This is even more the case with today's complex neural network generators. The Deep Dream neural network image generator [240] was a precursor to today's transformers, GANs, and diffusion models. Since it used classifications from the 2012 ImageNet image recognition dataset, the disproportionate number of dog-breed categories affected the aesthetic results of the generator [69], which is sometimes referred to as the "puppyslug" effect (Fig. 5.8).

This should not be surprising: in film the meaning derived from the juxtaposition of two intercut images, referred to as the Kuleshov effect [202, p.5], works as a part of film language regardless of the content it is used with, but the meaning of a particular montage depends on the content being cut. The poetics of procedural generation exhibit analogous properties: the content used in the generator matters.[18]

Many generators produce very different results depending on the data that they use. *Spelunky* [213] uses different templates for each area (Jungle, Ice World, etc.)

[18]This is, notably, a point Kuleshov might disagree with: his original formalist point was that the content was subsumed by the form, influenced by the scientific management techniques of Taylor [276]. Though Kuleshov's later writings revised his view to emphasize that, "In no case should one assume the entire matter of cinematography to be in montage" [202, p.195].

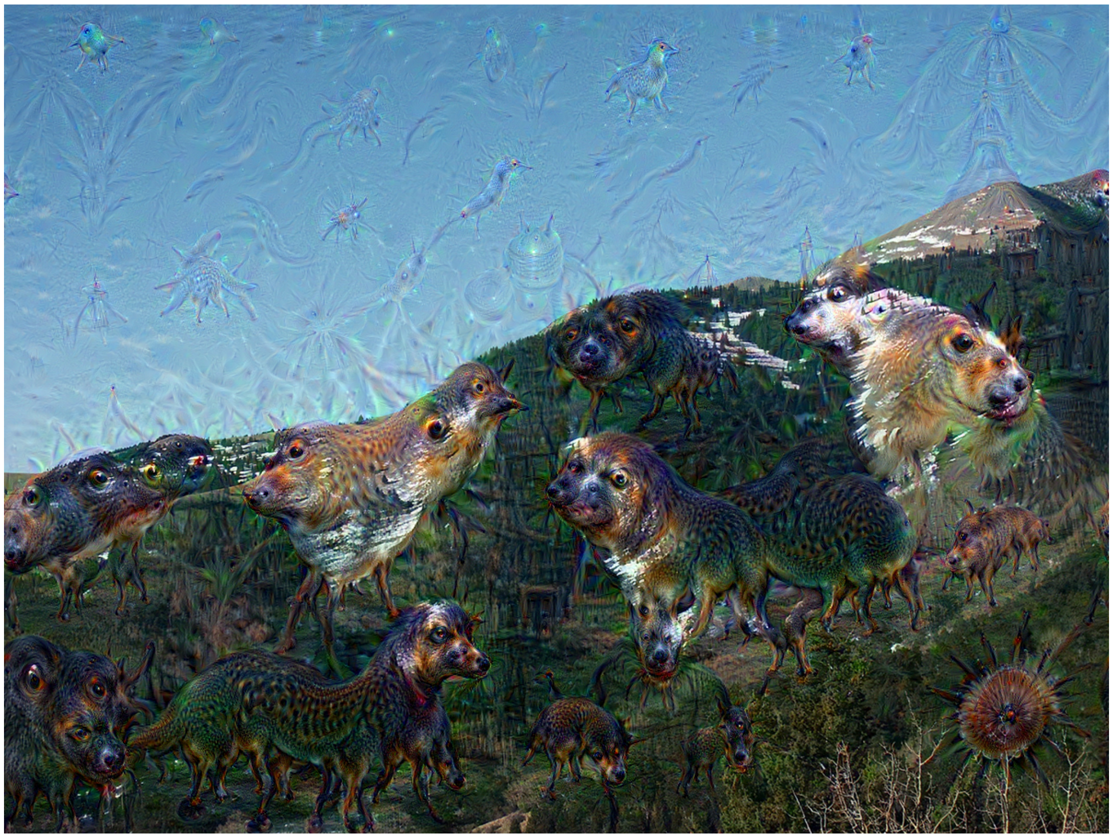


Figure 5.8: Puppyslugs: A photograph after running Deep Dream. Note the animal shapes: the initial Deep Dream model used the 2012 dataset from the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) [298], which included “fine-grained classification on 100+ dog categories” as an additional task. Image generated by the author.

and room type in its level generator. *Dungeon Crawl Stone Soup* [154] has a large library of vault templates that it similarly deploys across its dungeon branches. *Dwarf Fortress* [22] can be modded to generate very different worlds simply by editing the data files it uses.

5.6.3.3 Metadata: Cohesion

The struggle with data is that it needs to be self-consistent enough for the generated content to be perceived as a unified, sensible result. Some forms of data are able to use implicit relationships, such as the relationship between points in Perlin noise [267], or the relationship between different instances of Perlin noise with similar parameters. But additional metadata is often needed to give the generative system enough context to assemble a coherent result.

Metadata can be as simple as a tagging system giving context to the strings in a text generator. And for many generators that is enough, when combined with connotation, allusion, and elision, to maintain the illusion of complex relationships (Dias, 2016). But, following the SimCity effect, where the expectations the surface creates connect to the procedural system they represent [370, p.301], the most effective metadata is visibly related to the shape of the underlying generative system.

Often the automated creation or use of metadata is an important step in automating the generator. The innovation of word vectors [218, 231] has simplified natural language processing in useful ways, giving us a rough but effective way to quantify the meanings of words. The image/caption relationships learned by CLIP can be used to guide many other machine learning applications [281].

5.6.4 Beeswax

When we analyze a generative system, we should also pay attention to how it integrates hand-authored content and how it intersects with the fixed systems that it lives between. For Emily Short, the use of corpora falls under the heading of Beeswax, drawing on the image of a hive sharing the task of constructing a honeycomb. Short points out that using a human-assembled corpus inevitably also brings in the cultural assumptions of those who compiled it [313], which is its own form of metadata.

Beeswax also includes the idea of one-off content and content that is "hand-written for particular cases" [311, p.90]. This point should not be overlooked: when discussing procedural generators, it is tempting to focus solely on the most generative parts, but the hand-authored content often anchors the generator's structure. NetHack [255] is well known for including hand-authored responses to unusual situations, inspiring the Usenet meme that "the dev team thinks of everything" [40].[19]

5.7 Going Beyond Basic Poetics

My primary intent behind presenting this framework is to further the development of the vocabulary we use to discuss procedural generation. The original version of this poetic framework was deliberately incomplete, to emphasize that there are aesthetic effects that I have not considered in this chapter but which definitely exist. Balance

[19]See the many rec.games.roguelike.nethack Usenet posts to this effect; for example [10]

and contrast are common aesthetic categories across many mediums, and they can be applied across the space of artifacts (the Gaussian curve in the roll of two or three dice, the many possible maps for a Real-Time Strategy game) or within an individual artifact (the floorplan of a particular dungeon level). Other aesthetic lenses might be equally fruitful; some research has already been done in this direction (e.g. [324, Sec. 5])

In particular, the effect of different forms of probabilistic distributions has a deep scope for future examination. The discussion of the Apollonian and the Dionysian in noise only scratched the surface of noise color, warping noise with noise, and effect of the perceptual uniqueness of landmarks in generative space.

It is my hope that this framework, and future investigations into other aesthetic lenses, will lead us toward better metrics to use with Expressive Range Analysis, particularly when tailored to the design goals of a particular generator. When combined with user studies on how players perceive the aesthetic effects of a generator, it might be possible to operationalize these dimensions as evaluation metrics, in which case the design and evaluation of generative methods can begin to develop meta-metrics.

5.7.1 Juxtaposition

A meta-theme across many of these aesthetic dimensions is being able to observe the difference *and* the similarity between artifacts. For example, the repetition/orchard dimension is about being able to compare small differences because the larger possible differences are held constant. The clusters of aesthetic axes are, in part, about differentiating between different ways to express what artifacts have in common with each other and where they differ. Many of the tensions described in this chapter get their effect, therefore, through juxtaposition. We can regard juxtaposition as the crucial hint: what part of the process that created these artifacts is significant?

5.8 Other Aesthetic Effects

The framework could be expanded to encompass more aesthetic effects that frequently occur with generativity. To the best of my knowledge, the list has no end but this chapter does, so I will merely briefly mention a few:

Numinousness, a sense of touching the divine, something larger than human comprehension. In generativity it is often closely associated with a sense of infinity: Borges often invokes this in his short stories this way, as in the "The Book of Sand" [32] or "The Library of Babel" [34]. Generativity is one of the few ways I know to reliably design this sensation.

Despair and ennui at the useless variation. In "The Library of Babel" [34] Borges mentions librarians driven to destructive despair by the sheer randomness; in "Funes the Memorious" a man who uselessly remembers every detail he has ever seen. This is caused by too much randomness; or too many artifacts for the amount of complexity in the generative system; or insufficient ritual in encountering the next artifact, which leaves it feeling cheap and disposable.

Serendipity, discovering something good you would never have encountered intentionally. Artists and other practitioners have often cultivated randomness and similar methods to break out of epistemic ruts. The Dada cut-up technique [47], the OuliPo speculations about combinatorics and the clinamen [48], musical dice games [152], the connection between procgen and bibliomancy [220], and other approaches all gesture towards how serendipity can be deployed aesthetically.[20]

Surprise is one of the oldest reasons to include procgen in a videogame. A desire for surprise is what motivated the developers of Rogue (page 75): after all, the developers also wanted to be able to play the game. While keeping the developers entertained is seldom a goal for today's videogames, the value of building a system that can surprise its creators remains as relevant as ever.

Community, in exploring a shared space. Many competitive single-player games these days feature a weekly challenge, where players start from the same seed. Or, as with *Noctis* [127], *Elite* [38], *Elite:Dangerous* [112], or *No Man's Sky* [153], there is one vast but deterministic galaxy to explore. *MirrorMoon EP* [284] takes this further and generates a galaxy-wide puzzle for players to collaborate on. *Desert Golfing* [28], with its unending but deterministic golf course, similarly invokes shared exploration.

Ownership, feeling an attachment to a generated artifact. The uniqueness of a generated artifact makes the player more attached to it, feeling like it is something special just for them. However, when a generator can produce a seemingly-infinite number of other, similar artifacts, this often requires some additional ritual or social practice to cement the sense of ownership. This should not be surprising because that is also how ownership works with real-world objects, of which there are a similar infinite

[20]As one example, serendipity was a major motivation for my own *Virgil's Commonplace Book* [181].

number. The crossover between ownership and procgen has been discussed in the context of casual creators [59, p.9–11,229–232], but many implications remain unexplored. (Contrast this with the discussion of disposability in Sec. 5.5.3.1)

History, a sense of an artifact having past attachments in the world. This is related to ownership, but extended to the context of the artifact. We have to do extra work for a digital generated artifact to give us this aesthetic effect: actual objects get this for free. The pattern of scratches on a real-world mass-produced object reveal it has a history, whereas the engravings in *Dwarf Fortress* have a sense of history because of the immense simulation effort. Nevertheless, because generated artifacts can be unique, they have an easier time acquiring history if we let them. And they can be generated with prior attachments, whether real or invented. And that implied causality is closely connected to having a sense of purpose. (Compare with the discussion of context in Sec. 5.6.3.3, and style in Sec. 5.5.3)

Purpose, the sense that an artifact exists for a reason. Again, we have to work for this, because many generative systems are black boxes prone to the Talespin Effect [370, p.143–147]. However, a generative system that is clear enough to give an artifact a sense of context for why it was generated (or at least the forces that operated on it) has an aesthetic effect that draws the player toward a deeper understanding of the system and the place that particular artifact occupies within it. (Compare with the discussion of structure and implied causes in Sec. 5.5.1.1)

Discovery, finding something in the generative possibility space. This is most consciously invoked by purely space exploration games such as *Noctis* [127] and its spiritual descendants such as *No Man's Sky* [153]. However, discovery is a common element across the board: finding your next interesting weapon—such as in *Dungeon Crawl* [154], *Diablo* [29], or *Borderlands* [212]—is an example of discovery in more abstract generative possibility spaces.

Causality can have aesthetic side effects. While we've been focused on the *why* rather than the *how*, the cause of the generator (Sec. 6.2.2.1) affects the results, specifically where the model breaks down in the sense Brian Eno describes [101, p.283]. The fashion in which the generator strains against its assumptions or breaks down in characteristic ways profoundly affects the player's experience and after lengthy interactions sometimes becomes the dominant aesthetic experience.[21]

Change is a subset of the generative gestalt aesthetic: instead of varying across artifacts, we vary one artifact over time. Time, however, has two additional effects: first, it creates a history and implies a cause-and-effect, which is a powerful thing to leverage in procgen, even when invented ex post facto [138]. Second, the comparisons between the same artifact at different points in time become more obvious, even more so than with the trees in an orchard. As long as the player accepts the implied continuity, the changes can be more extreme while still being read as related differences.

Exploring these in more detail, and listing additional new aesthetic effects, would be a valuable contribution to our understanding of generativity.

[21]We might term this the Battler/Erica distinction (cf. [185]): is the player being drawn deeper into thinking about the signified thing that the model represents, or do they lose sight of that and concentrate on signifier to the exclusion of the signified? Are we trying to solve the mystery to deconstruct it into its constituent parts, or are we trying to understand the message the storyteller is trying to convey? This parallels Mitchell et al.'s concept of *storygameness* [237], suggesting an interesting affinity between the space of a story volume and the generative possibility space of a procgen generative system.

5.9 Conclusion

I have described a series of lenses for analyzing procedural generation from an aesthetic and experiential perspective (Table 5.1). This fills a critical gap in the discussion: the lack of vocabulary to talk about the *why* of procedural generation has led to inappropriate application of metrics—for example, the popular press gushing about use of 64-bit seeds in *No Man's Sky*, or applying the linearity and leniency metrics introduced in expressive range analysis [326] to generators that have nothing to do with *Mario*-style level generation.

Using the framework presented in this article, game studies critics have a more nuanced way to discuss the output of generative systems, designers and developers have tools to describe their priorities when designing new generative systems, and researchers have a blueprint for defining their research into new forms of procedural generation.

The way a generative system expresses **complexity** and information can be through orderly **Apollonian** patterns, **Dionysian** noise, or complex combinations of the two. The **form** of the generative system can expressively use **gestalt** forests, **repetition** of orchards, or the generation of **individual** trees. The **locus** of the generation can be on the **surface** details of an individual artifact, as in the map in *Exile*; the **structure** of the generation process, as in the pattern of diamonds in *Minecraft*; or the **gestalt** of the generative system's output, as in the planets in *No Man's Sky*. Individual artifacts and gestalt impressions bridge both the form and the locus. Orthogonal to both form and locus, the **variation** can exhibit the **multiplicity** of perceptually unique output, **cohesion** of the results, and a **style** that conforms to a goal. The different principles are related but distinct, perhaps best imagined as a multidimensional vector with partially interrelated axes, or a series of contrasts and congruencies.

In this chapter, I presented a poetics of procedural generation in games and generated novels, sketching out some of the principles that describe how generative systems are used and how we can meaningfully discuss the shape of the things they generate. It is my hope that this will inspire further refinements and foster deeper discussion of how procedural generation is used in games.

5.9.1 Practical Questions

A useful practical application of the framework is to look at a generative system with questions for each of the aesthetic dimensions.

Individual What effect does a single artifact have? What are the differences between artifacts? What makes this artifact stand out?

Gestalt What is the overall impression of the generative space? What is the effect of the group of artifacts?

Repetition What are the similarities between artifacts? What does the player learn about the generative system from their commonalities?

Structure What does this artifact tell us about the generative space? What processes can we discern? What can we predict about the next artifact we see?

Surface What direct impact does this artifact have on our experience? What details do we pay attention to?

Multiplicity How much variance can we perceive across the generative space?

Cohesion Do the artifacts feel like they belong? Do they fit in with the other artifacts? Can we perceive them as part of a cohesive system?

Style How well do the artifacts fit their context? Can the generative system adjust to match the current needs?

5.10 Acknowledgements

This chapter uses material originally included in "Preliminary Poetics of Procedural Generation in Games" [183], which is indebted to the guidance of Noah Wardrip-Fruin and Adam M. Smith, and extensive arguments with Max Kreminski. Additionally, I would like to thank Batu Aytemiz, Jesse Harder, Nic Junius, Oleksandra Kheel, Max Kreminski, Stella Mazeika, Joseph Osborn, and Xiaoxuan Zhang for providing valuable feedback along the way.

Chapter 6

Generators that Read

Recall the second research question of this dissertation, about the lower-level generative operations that make up a generative system. (RQ2) Of the different categories of generative operations, perhaps one of the least studied is reading (Sec. 8.1.2). While many generators operate over simple inputs, or merely unfold autonomously from a single seed, in recent times there has been the development of interest in forms of generative systems that interpret complex forms for input and meaningfully respond to them. Often, these inputs are things that were not originally intended to be used as input to a generator. Challenges such as the Settlement Generation Challenge of the Generative Design in Minecraft Competition [301] have explicitly encouraged a focus on the development of *context-sensitive* generators, capable of taking an arbitrary *Minecraft* map as input and generating a settlement that fits well within the context of that particular map. Projects like *WikiMystery* [20] have used existing corpuses of open data as a foundation for the generation of murder mystery scenarios. And essentially all approaches that fall under the umbrella of *procedural content generation via machine learning* [344] begin by training on a large corpus of input data.

Nevertheless, there remains a tendency in the community to talk about generation as though it is primarily a process of *writing* or ex nihilo creation of artifacts, sidelining or even outright erasing the sophistication of the increasingly complex components of generators that concern themselves primarily with the *reading* of input. I believe it is worthwhile to look more closely at how generators read.

At the same time, there is a tendency within computer science research to place an emphasis on *correctness* and *unambiguousness* in the development of algorithms for interpreting complex forms of input. From natural language processing to emotion recognition from face images, much of the literature implicitly assumes that it is both possible and desirable to produce an objectively correct and unambiguous interpretation of these complex inputs within the computer. I contend, however, that extracting machine-usable meaning from complex input necessarily entails a creative act of interpretation: the complexity of the input ensures that it could always be read differently, and the approach to interpretation that you choose to apply will affect the nature of the interpretation you produce.

Moreover, I argue that, for the purpose of procedural content generation, it is often beneficial to accept “incorrect” and “ambiguous” readings of complex input as valid, enabling our generators to produce surprising outputs by selecting from among a wide range of mutually incompatible but equally viable interpretations of the same input. Likewise, when designing generators to be used in mixed-initiative contexts, it may be desirable to embrace multivocality by exposing the user to a variety of possible interpretations of the same input, thereby helping them to see alternatives they might not otherwise have considered.

There are many different approaches to reading, and within the humanities, many forms of critical practice are organized around a particular theory or methodology of reading. Borrowing this lens, I suggest that different approaches to reading might inform or inspire the development of novel approaches to procedural generation and that analyses of existing generators may benefit from thorough examination of how these generators read their input in a way that engages deeply with a particular theory of reading.

Rather than viewing generators as monolithic black boxes, generators can be viewed as a pipeline of data transformations [64]. Many common procedural content generation techniques can be subdivided into modular units that chain into each other, often forming a complex web (Chap. 8). In some fields, such as parametric 3D modeling or shaders, these connections are made explicit through the use of node-based interfaces (Sec. 4.2). Node-based interfaces explicitly route the outputs of one stage of the pipeline into the inputs of the next (Fig. 6.1). In the PCG field, this has been expanded by the Generominos ideation cards, which model a pipeline of data transformations, explicitly highlighting the importance of matching inputs and outputs [65].

For example, consider WaveFunctionCollapse (WFC) [143], a constraint-based and example-driven approach to procedural content generation. WFC is composed of two linked generative processes: an input model that translates an image into adjacency

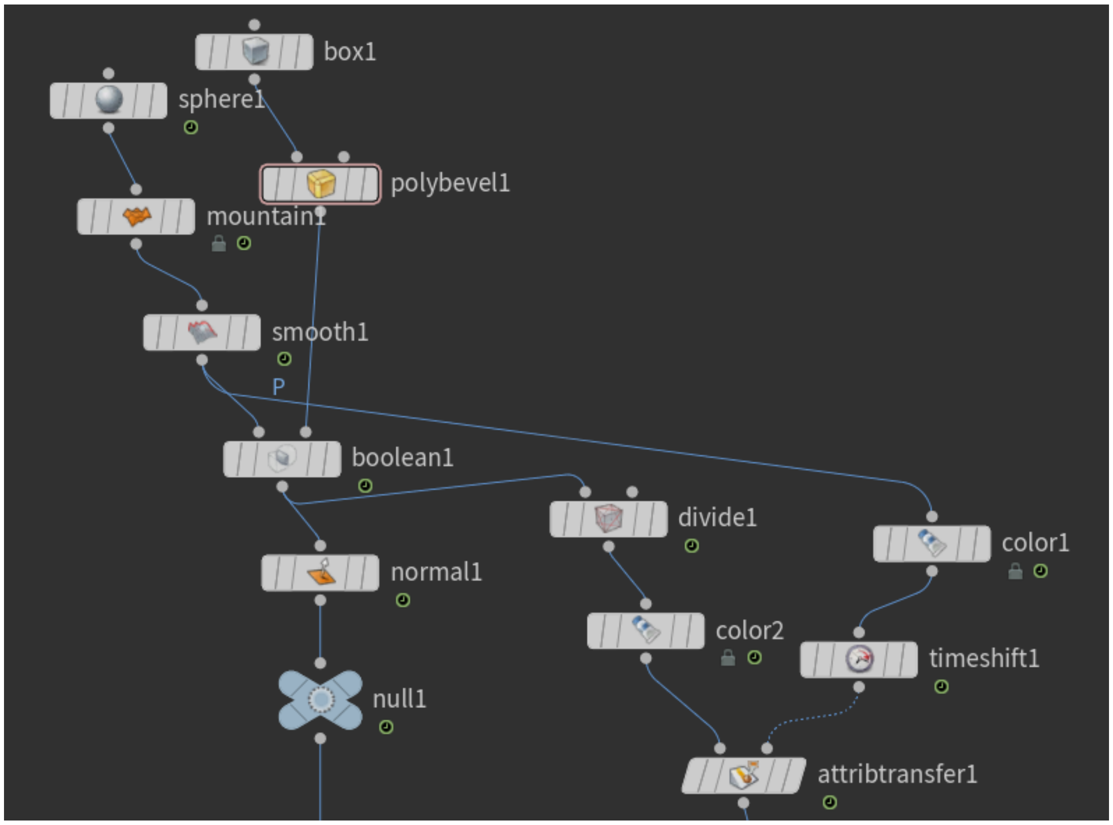


Figure 6.1: A portion of a node-graph in SideFX's Houdini, a procedural 3D modeling and effects software, demonstrating how the web of nodes is connected via inputs and outputs.

constraint data and a probabilistic constraint solver that turns the data into new generated images [186]. The constraint solver cannot act on its own without some form of input to specify the constraint data. The generator needs both its input and its output components to operate. Further, WaveFunctionCollapse has multiple input models that read input data with different approaches. By examining how the different input models process the data that the generator reads, we can gain greater insight into the process of the generator as a whole.

As another example, consider that many of the novel generators created for National Novel Generation Month (NaNoGenMo)[1] draw on the same source texts—frequently including *Alice in Wonderland*, *Moby-Dick*, and *The Odyssey*—yet produce very different outputs. This is possible in part because each generator takes a different approach to reading its input text.

In this chapter, I focus my investigation primarily on generators that read complex inputs which were not originally intended primarily as inputs to a generator. A generator that reads complicated generator-specific configuration files, for instance, is of less interest than a generator that reads *Minecraft* worlds or arbitrary English texts. Nevertheless, generators that read complicated forms of generator-specific input may still be amenable to some of the same forms of analysis, so I do not exclude them entirely from the scope of interest.

It is important to note that none of the examples in this chapter are intended to suggest that reading can be reduced to simply statistically modelling a text. A statistical

[1]`https://nanogenmo.github.io/`

model might be one way to approach a distant reading of a text, but in isolation it fails to capture the full possibilities of these humanities-inspired reading lenses.

6.1 Why Do We Want Our Generators to Read?

6.1.1 Context-Sensitive Generation

There are a wide range of problem domains that call for *context-sensitive* forms of procedural content generation: the generation of artifacts that, rather than standing alone, are expected to fit in to some existing complex context. One example of a problem that calls for context-sensitive generation can be found in the Settlement Generation Challenge of the Generative Design in Minecraft Competition [301], which tasks competitors with building a generator that can produce a convincing settlement on any arbitrary chunk of terrain in the voxel-based construction game *Minecraft*. In order to reward competitors for producing generators that are truly context-sensitive, competitors are not given access to the specific maps that will be used as testbeds for their settlement generation processes. Therefore, they must do their best to produce generators that are capable of functioning in a wide range of potential contexts, and are disincentivized to produce generators that are prone to overwriting large swaths of the existing terrain without regard for how a generated settlement might fit more naturally into its surroundings.

In other cases, it is often desirable to generate content that fits around or fills the gaps between a number of fixed "landmarks" without overwriting those landmarks. This too necessitates generators that are capable of reading and responding to an established context.

6.1.2 Generative Pipelines

Building on the notion of context-sensitive generation, it is important to acknowledge that many generators do not operate in a vacuum. In particular, especially in games that make extensive use of procedural content generation, a single generator is often merely one component of a larger generative pipeline that consists of many generators wired together end-to-end. In these situations, the complex output of one generator becomes complex input to another generator, and the downstream generator must then interpret the input in some nontrivial way in order to generate an output artifact that matches or meaningfully adapts to the input artifact.

When working with procedurally-generated base terrain, a frequent problem is appropriately respecting the elevation of the terrain when placing objects, particularly when generating buildings, towns, and road networks on rough terrain [100]. For example, *Minecraft* settlement generation is itself an instance of a problem where generative pipelining leads to a need for downstream generators that are capable of interpreting and responding to the complex output of upstream generators (in this case, the base terrain generator itself) [301]. Likewise, world generation in the roguelike *Caves of Qud* makes use of multiple distinct generators, each of which feeds into other generators in the pipeline [137].

6.1.3 Mixed-Initiative Co-Creativity

When building mixed-initiative co-creative tools [384] that attempt to use procedural generation to supplement or augment the work of a human user, it is especially critical for the generative systems employed by the tool to be capable of reading or interpreting whatever the human user has created so far. In cases where the generator is not capable of doing this, it is likely to step on the user's toes in various ways, for instance by overwriting their work and replacing it with generated content.

This behavior can be seen in the context of mixed-initiative 2D platformer level design tools with *Morai Maker* [146], an AI-driven game level editor in which a human user and an AI level designer take turns collaborating on a single shared design. Unlike earlier human/AI collaborative level design tools such as *Tanagra* [327], which provides the human user with a suite of tools for communicating their design intent to the AI directly, *Morai Maker* attempts to infer what it should do in response to the human user's actions largely without explicit guidance. Partly as a result of this lack of guidance, the AI collaborator has a tendency to apparently ignore or repeatedly overwrite the human user's edits to the shared design, which can produce frustration in users who desire a greater degree of control over the design process. The user experience of *Morai Maker* under its current design constraints, then, hinges on its ability to read the design the user has created so far, ideally with an eye to deriving an understanding of the user's intention purely from the actions they have taken. Improving the AI collaborator's capability to read the human user's partial level designs would directly result in an overall improvement to the user experience of collaborating with the AI.

6.2 What Does It Mean to Read?

There are many different kinds of reading. Within the humanities, the term "reading" has taken on an expansive definition as an umbrella term under which a wide variety of approaches to the analysis and interpretation of texts may be considered. Indeed, following the *linguistic turn* [290] in the humanities, the term "text" has itself taken on a broader meaning than in its original sense of purely linguistic or written works, and is now widely understood to encompass all kinds of cultural artifacts [107, 215], from advertisements to zoo signage. As such, the notion of "reading" is a contested one, and merits further examination if we are to apply it as a lens to the understanding of generative methods. In this chapter, I do not attempt a comprehensive survey of all possible approaches to reading, as such an undertaking would be well outside the scope of this work. Instead, I offer samples of several diverse perspectives on the question of what it means to read, with the goal of illustrating the range of approaches that are possible and hinting at how different approaches to reading might inform or inspire different approaches to procedural generation.

6.2.1 Close Reading

Close reading is "the thorough interpretation of a text passage by the determination of central themes and the analysis of their development" [171]. "Close reading concerns close attention to textual details with respect to elements such as setting, characterization, point of view, figuration, diction, rhetorical style, tone, rhythm, plot, and allusion," often examining the gap between what is said and what can be inferred [286]. The methodology is evaluated, in part, by its explanatory power for the details of the presentation.

A generator that performs a close reading of its input is concerned with the details of the input. To give one example, Interactive Data Visualization Inc.'s SpeedTree middleware creates trees and other vegetation for games and visual effects. The generator can take into account the shape of the nearby terrain—at a fairly high level of granularity—when placing a tree. The generator must read the context it is placing the generative artifact into.

6.2.2 Distant Reading

Rhetorically positioned in contrast to close reading, in *distant reading* "the reality of the text undergoes a process of deliberate reduction and abstraction" [241]. Rather than concerning itself with the details of presentation, distant reading is a process that operates on models and visualizations of the text. This thousand-foot view reveals commonalities and structures that would otherwise go unseen but that can now be visualized by graphs, maps, trees, and other data structures.

One approach to designing a generator is to program a model of the process that created the desired result. There's an existing term for this in generative methods: a *teleological* generator [17], where the generator models the process that created the phenomenon. In Barr's terminology, a teleological model "incorporates time-dependent goals of behavior or 'purpose' as the primary abstraction and representation of what an object is" [17]. Barr's category of the teleological model that emulated the physical processes of the system was contrasted against the traditional method in computer graphics of directly specifying an object's shape and behavior.

Another approach is to generate something that looks like the result we want, without caring how we get there. Fractals often look like things (mountains, tree bark, clouds) but the process of producing a fractal is very different from the process of producing a mountain. The late Ken Musgrave termed this "ontogenetic modeling" [254, p.442–444].

A teleological terrain generator [17] might include simulations of geological processes, erosion, the shifting flow of rivers, and so on. In contrast, an ontogenetic terrain generator [254] might use Perlin noise to emulate the shape of the desired terrain. In either case, the generator is modeling a system. Creating and analyzing the generator both involve a process of reading via that model.

6.2.2.1 Causes in Generativity

This terminology is less than ideal. The existence of constraint solving and search-based machine-learning techniques complicates this terminology: teleology in the general sense deals with the end purpose of something, rather than the efficient cause of how it was made [105]. A generator that starts with an end goal in mind and works backwards to find a solution for it (e.g. a constraint solver) seems more properly fitting for the term.

"Ontogenetic" is also a bit confusing because while ontology is about the morphological characteristics of an organism it is also concerned with the process through which an organism grows and develops, which does suggest some relationship between the process and the thing it is modeling.

We can categorize generators based on causation:[2]

A **material-cause** generator directly creates the artifact, for example inserting it from a template. You can make a mountain by taking GIS heightmap data and directly stamping it down into the terrain heightmap. This can be compared with the traditional computer graphics modeling approach of directly specifying geometry.

A **formal-cause generator**, in the sense of a platonic form, is concerned with having a process that produces something with the semblance of the desired form, without caring how it arrives at it: Perlin noise mountains, for example. Generating a narrative based on a non-narrative process is another example (e.g. a story generated from a chess game).

An **efficient-cause** generator cares about how something was made, with the intent to physically model the process: creating mountains by simulating tectonics and erosion.

A **final-cause generator** doesn't care about the process at all but does care about the results: we need the mountain to include these particular paths and look like a mountain, but the constraint solver is free to make up the rest.

While each of these approaches might create identical-looking outputs, the

[2]We should, however, be careful: while this kind of distinction can be useful, it skirts up against the labeling problem (Sec. 8.6).

edge-cases, failure-modes, areas of flexibility, and relationship with other content can be very different. Each has its own characteristic failure mode, in the Brian Eno sense [101, p.283].

All computational models are at best abstracted representations of the things they simulate, and generative systems are no different. This can be deliberately leveraged by creating a juxtaposition between the artifact and the process that created it (cf. Sec. 5.7.1).

A useful concept here is *mimesis*: the representation of the real world in art. While many generative systems try to be mimetic, we need to recognize that they are signifiers, not the signified: a generative system can *represent* something, but it fundamentally isn't just *about* that thing (cf. Sec. 8.6). In fact, a non-mimetic system can be a powerful form of procedural rhetoric: by associating *mimesis of artifact* with an alien process, it can create a *computational caricature* that implicitly makes a statement about what aspects are important. A *computational caricature* operates by exaggerating the unique features of a process, making them interpretable, which is one reason they are useful for design research [318, 322].[3]

6.2.3 Critical Approach

A critical approach to reading is performed by mapping a theory onto a literary work to explain its meaning, a two-directional process where "the theory should

[3]Some terminology that might be useful for discussing this mimesis was suggested in a conversation with Max Kreminski: A mountain that is generated through tectonic simulation is using *mimesis of process*. A mountain that is generated through Perlin noise has *mimesis of artifact* but is using a non-mimetic process.

illuminate a work, and a work should illuminate a theory" [286]. We can characterize image generation via deep learning neural networks (such as Deep Dream [240]) as a generator that reads its input by mapping a theory (learned in training) onto the input image.

6.2.4 Hermeneutics

A *grammatical hermeneutic* reading attempts to derive the meaning of a text through analysis of elements that are present within the text itself, rather than from elements outside the text, such as the intention of the author [371]. For our purposes, an important factor to recognize is that this often deliberately results in multiple parallel readings of a single text. For example, the European medieval exegesis of sacred texts, influenced by Aquinas, simultaneously looked for four senses in every text: a literal (*sensus literalis*); moral or tropological (*sensus tropologicus*); allegorical (*sensus allegoricus*); and mystic (*sensus anagogicus*) sense [50] [158, p. 99].

The recognition that a single work can have multiple senses challenges the assumption that a reading will arrive at a single, unambiguous classification. A text can be read in multiple ways simultaneously, and multiple generators can produce a variety of valid interpretations—even mutually incompatible interpretations—of the same input.

6.2.5 Poetics

In contrast to hermeneutic approaches to reading, poetics represents an alternative perspective that focuses instead on the *felt effects* of a text in the reader [80,288].

Whereas it is fairly straightforward to see how close or distant reading might be employed in the construction of a generator, it is less obvious how the framework of poetics might be applied to a machine reader, especially insofar as the term “felt effects” may be interpreted as concerning itself primarily with a text’s *physiological* effects. Nevertheless, generation based on a subjective experience of a text—perhaps from the perspective of one interpreting agent among many in an artificial artist commune such as CheapArtistsDoneQuick [67] or The Digital Clockwork Muse [304]—remains an intriguing possibility.

For an example of an existing generative technique that may exhibit something like mechanical felt effects, consider word vectors, which are constructed through a mechanical analysis of word adjacencies [232,233]. Because word vectors describe points in a much larger *continuous* space, they allow for a kind of “semantic bleed” between adjacent words, similar to how ambiguity and wordplay operates in textual poetics for human readers. Further, word vectors capture semantic relationships in the text that was read [234], thus indirectly modeling or mimicking some of the subjective effects of reading in human readers through their very method of construction.

6.2.6 Proceduralist Readings

Proceduralist readings represent still another approach to reading, this time an approach native to game studies and focused primarily on the interpretation of interactive or rule-based texts such as videogames. Proceduralist readings “address a convergence point between [...] expression and interpretation” and focus on “internal readings of a game's dynamic" yielding "meaning derivations" [358]. These meaning derivations are structured logical arguments for the interpretation of the game's mechanical and sensory cues as the higher-level meanings that emerge from the game's dynamics and aesthetics.

Proceduralist readings have themselves been proceduralized: building on the Operational Logics framework as a game description language, Martens et al. describe a procedure for automated reasoning about games [222]. This proceduralization, in turn, has become an essential component of Gemini [342], a generator of abstract games. Gemini provides users with a specification language that they may use to specify what arguments they want the generated games to make. Gemini then uses this specification to guide its search within the design space of possible games, identifying and returning games that may be read in the desired ways.

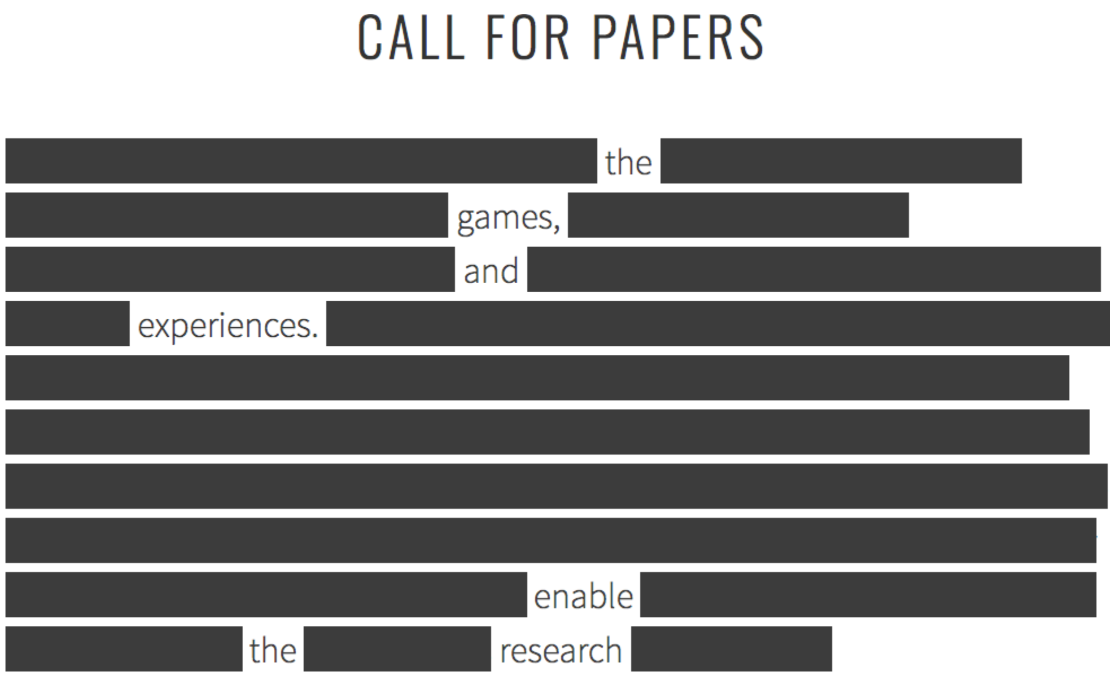
6.3 Case Study

As an illustration of the potential importance of reading to the generative process, I now present a brief comparison of two similar erasure poetry generators: *The Deletionist* [36] and *blackout* [196]. Both of these generators are packaged as browser bookmarklets and present themselves as ways of turning arbitrary web pages into poetry by erasing most of a page's text. Moreover, the processes by which these generators write their modifications to the target page are both straightforward and nearly identical. The difference between these two generators thus lies almost entirely in how each generator reads or interprets a page's text prior to modification.

The Deletionist interprets each webpage as a single unit, considering all the text on the page at once rather than breaking it up into smaller pieces for analysis. It decides which words to erase deterministically, such that running it repeatedly on the same webpage will produce the same result every time. Its selection of which parts of the text to retain is based on one of several regular expression patterns, many of which use either the start or end of words to determine whether some or part of the word should be retained. In some cases, for instance, it will choose to retain primarily words beginning with the letter M, while in other cases, it will choose to retain words ending with a period. It also frequently retains certain common whitelisted words, such as "from" and "like". Once it has decided which parts of the text to retain, all other parts are erased.

blackout takes a markedly different approach. Rather than treating the whole page as a single unit, it reads each paragraph in isolation and makes no attempt to coordinate its reading of one paragraph with its reading of the next. It reads non-deterministically, such that running it repeatedly on the same webpage will typically produce different results from one run to the next. Its selection of which words to retain, meanwhile, makes use of part-of-speech tagging and probabilistic fuzzy matching of valid sequences of parts of speech, recognizing simple declarative sentences that could be formed by omitting some or all of the words in a paragraph and selecting some valid sentence for each paragraph.

A side-by-side comparison of the outputs that these two generators produce

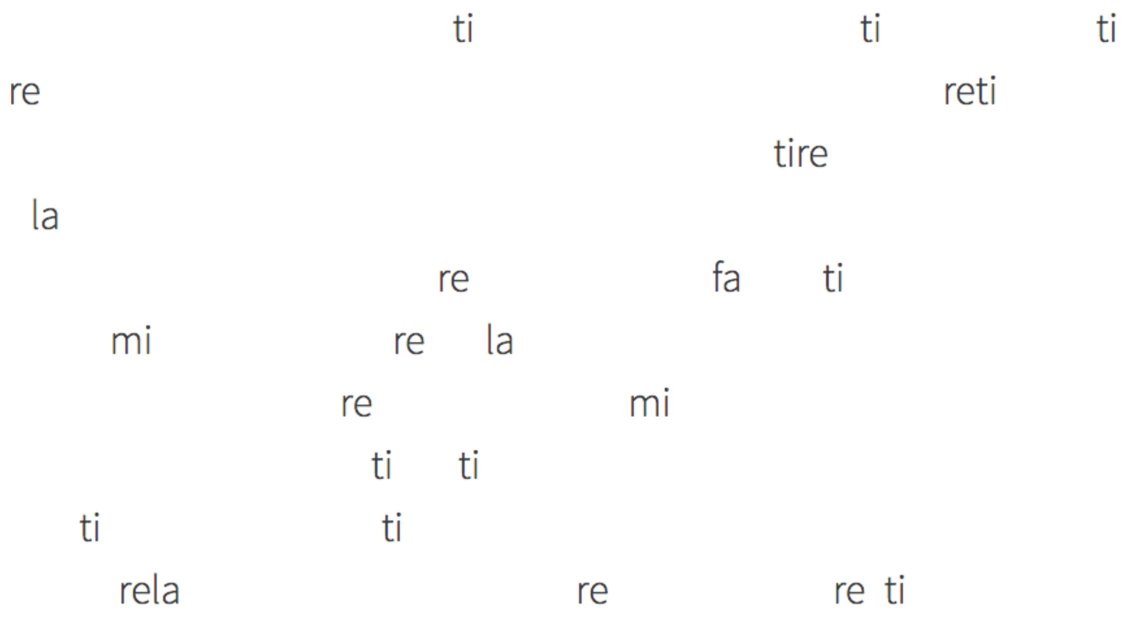

CALL FOR PAPERS

the
games,
and
experiences.
enable
the research

Figure 6.2: An erasure poem generated by running *blackout* on one paragraph of the 2019 PCG Workshop call for papers, taking the form of a simple declarative sentence.

when run on the same input page will further confirm that the differences between them lie largely in terms of how they read the input they are provided. This serves to illustrate an important principle: for generators that take nontrivial input (and that can thus can meaningfully be said to consist of a distinct “reading” and “writing” component), it is possible to alter or replace either the reading or the writing part without changing the other component and still get interestingly different results.

Both generators are equipped with a variety of patterns, each of which is tested against the input before the generator makes a final determination about how to generate the output poem. This can be seen as leveraging a form of internal multivocality, embracing the ambiguity of the reading process by engaging with a variety of interpre-



ti ti ti
re reti
tire
la
re fa ti
mi re la
re mi
ti ti
ti ti
rela re re ti

Figure 6.3: An erasure poem generated by running *The Deletionist* on the same input, taking the form of a series of syllables corresponding to musical notes.

tations of the same input. Moreover, in the case of *blackout*, the generator creatively misapplies a part-of-speech tagging algorithm taken from a natural language processing package to deliberately preserve ambiguities of interpretation in the source text. Rather than flattening ambiguous words (which could be parsed as having several distinct and mutually incompatible parts of speech) into a single most likely interpretation, as in the typical application of similar algorithms, *blackout* avoids this flattening by treating words as having arbitrarily many distinct part-of-speech tags until the final poem is rendered.

6.3.1 Conclusion

As evidenced by the brief sampling here, a wide variety of theories of reading have been introduced, and each such theory has interesting potential implications for generators that read. Proceduralizing various approaches to reading may prove a successful strategy for the discovery of new approaches to generation. Moreover, deep engagement with a particular theory of reading may enable deeper analysis of existing generators that process complex inputs.

As a brief illustration, I contrasted two erasure poetry generators, *The Deletionist* [36] and *blackout* [196] and demonstrated that the differences between them rests almost entirely on the different approaches to reading the input.

Many generators have interesting approaches to reading. However, the way generators read is less discussed than how they write, and seldom separated into a subject worthy of discussion on its own. Despite this, I have demonstrated several ways in which particular generators cannot be understood without first examining how they read. Therefore, when we discuss generators, we should go beyond discussing what the output looks like and consider including a clear separation in our discussion: between the ways in which a generator writes and the ways in which it reads.

By treating the problem of extracting machine-usable meaning from complex input as a form of reading, I suggest that—when crafting a generator that reads—it is often desirable to preserve ambiguity and embrace the possibility of incorrectness, rather than attempting to read correctly and unambiguously. Interpretation is essentially a kind of creative act, and different approaches to interpretation may inform the development of new approaches to generation. Different ways of reading can enable different kinds of generation and point to methods that can inspire new approaches to generation. A terrain generator that models a physical process reads its model in a different way than a tree generator that curves roots around nearby stones. The idea of a proceduralist reading is an already accepted methodology in game studies, and, in part, inspires the introduction of generativist readings, as discussed in the next chapter.

It is important to note the double meaning of the term “reading” as it is commonly understood: the same term applies both to the process of interpretation and to the concrete interpretations that are produced through the application of this process. Reading a text produces a particular reading of the text in question, and a reading of a text may be examined, understood, or interpreted as a concrete artifact or text in and of itself. Therefore, when a generator reads an input text, it may be useful to consider the reading it produces as an artifact that merits examination, even if this reading is not intended to be directly consumed or experienced by the generator’s audience at the end of the generative process. In the following chapter, I further examine the implications of this view.

6.4 Acknowledgements

This chapter uses material originally included in “Generators That Read” [200], which I co-wrote with Max Kreminski and Noah Wardrip-Fruin. It was significantly

improved based on feedback from anonymous reviewers during the peer-review process, and the parts of the present chapter have been further developed based on subsequent conversations (particularly comments by Ian Horswill, later conversations with Kate Compton, and my dissertation committee).

Chapter 7

Generativist Readings

I originally developed the concept of *generativist reading* in close conjunction with Max Kreminski [200]. A generativist reading is an interpretation of a text into a generator of similar texts. (RQ2)

This can be compared with the concept of proceduralist readings [358], which established a methodology for deriving meaning through analysis of a game through the process of *meaning derivation*:

> After defining the components of internal interpretation, we will present a framework for meaning derivations. A meaning derivation is a hierarchical, structured "proof" for what a game means and is the method for a proceduralist reading. The point here is not to say that meaning can be objectively proved, but instead to compensate for the lack of attention to detail in the current state of videogame interpretation. In a meaning derivation, all assumptions of the interpreter are broken into very small units and then logically constructed into rigorous cases for a claimed meaning. [358]

However, in contrast to how proceduralist readings are applied to games, a generativist reading takes an artifact and imagines how it might have been created by a generative system. A proceduralist reading yields an interpretation. A generativist reading is also an interpretation of a text. Much like a proceduralist reading of an interactive text focuses on deriving meaning from the rules or procedures within the text, a generativist reading attempts to answer the question of *what this text can tell us about how to produce more similar texts.*

Generativist readings are part of the existing folk practices of generative system designers. Oftentimes, when we create a generative system to produce types of artifacts that were previously exclusively handmade, we essentially find ourselves manually conducting a generativist reading of a corpus of examples. For instance, if a human reader was to read *Moby-Dick* and handcraft a Tracery [63] grammar that utilizes vocabulary and sentence structures drawn from the book to produce sentences that sound plausibly as though they could be drawn directly from the source text, the resulting grammar would constitute a generativist reading of the text. Similar practices are not uncommon among Twitter bot creators, who may often begin by writing out the source text that a generator will try to imitate and then recursively substitute grammar rules in place of concrete words, gradually sublimating the text itself into a statistical model of the text. Manual generativist readings may even be used as an instrument of critique: consider Umberto Eco's proposal of algorithms for plot generation in the style of various filmmakers as a way of parodying those filmmakers' styles [97].

Performing a generativist reading of a text means that we have internalized a model of that text that is robust enough to reproduce that text. It does not mean that we have discovered the original method of creation: generativist reading does not recover original causes (cf. Sec. 6.2.2.1). It does, however, mean that the reading is making a statement about the meaning of the text by replicating a possible causal relationship.

A reading that properly matches the text is an interpretation of the text. It may approximate the text arbitrarily closely, but we should remember that it does not have objective value on its own. We could, in fact, deploy multiple generativist readings as a hermeneutical array, giving us multiple lenses on the text.

7.1 Mechanizing Generativist Readings

Machines can conduct proceduralist readings as Martens et al. demonstrates [222]. Generativist readings can likewise be mechanized. Mechanical processes of generation that rely on generativist readings typically begin by conducting one or more generativist readings of an input text or corpus. The generator then queries or manipulates these readings to produce individual output artifacts. For instance, text generation with Markov chains follows a two-step process. First, the computer conducts a generativist reading of the source text by moving over the text and tracking the overall frequency with which each word it encounters follows each other word. Then, the process of generation employs the statistical model created through reading to write new texts that imitate the read text. This same structure can be observed in many forms of generation, especially in procedural content generation via machine learning [344]—which hinges entirely on the construction of generative models from which individual output artifacts can then be sampled. Many filtering operations (Sec. 8.1.3) incorporate a reading process as part of their evaluation.

There are parallels between the manual enactment of a generativist reading and a mechanized generativist reading. One process for constructing a generative grammar (perhaps for the Tracery bot mentioned above) is to write out the source text that the generator will imitate and then recursively substitute grammar rules in the place of the concrete words, sublimating the text into a statistical model of the text. A straightforward mechanization of this is constructing Markov chains by reading the frequency with which one word follows another. The generator then uses the statistical model created from the reading to write new texts that imitate the read text.

7.1.1 Reading Operations and Generativist Readings

While reading operations (Sec. 8.1.2) and generativist readings both make use of reading processes, there are some significant differences. The first is that most reading operations take very complex input and output a much simpler representation metric. In contrast, the output of a generativist reading is an entire generative system.

While the input to a generativist reading is often complex, its output can be more complex. After all, the lower bound of what it can produce is a generative system that can reproduce the input artifact. A generativist reading, therefore, is most frequently classified as a generator transformation, not a reading transformation. In a sense, what we are doing when we perform a generativist reading is not a reading process, it is instead a generator process, at least in the technical sense of reading transformation operations. From the one, we get many.

Like other generative operations that output generative systems, generativist readings are therefore potentially part of a meta-generative topology. Filtering operations can change the generative space, rather than just individual artifacts. When including a generativist reading as an operation in a generative system, similar considerations apply. A generated generative system creates its own generative space, more or less by definition.

7.2 Generativist Reading in Practice

Some practitioners in the generative art world recognize the reading process as an intrinsic part of generation. For example, in the view of everest pipkin [272]:

> When I say that the creative act is the reader's, I imply the creator as well as the audience. When working with generative text, it is impossible not to read. One has to look for bodies of text that can function as useful sources for tools; big enough, or concrete enough, or with the right type of repetitive structure; learnable. And then one has to read the output of such machines, refining rules and structures to fix anything that breaks that aura of the space one is looking for. In this, we are not unlike the medieval scholar who studies holy verse to become fluent enough in that space that it becomes building block.

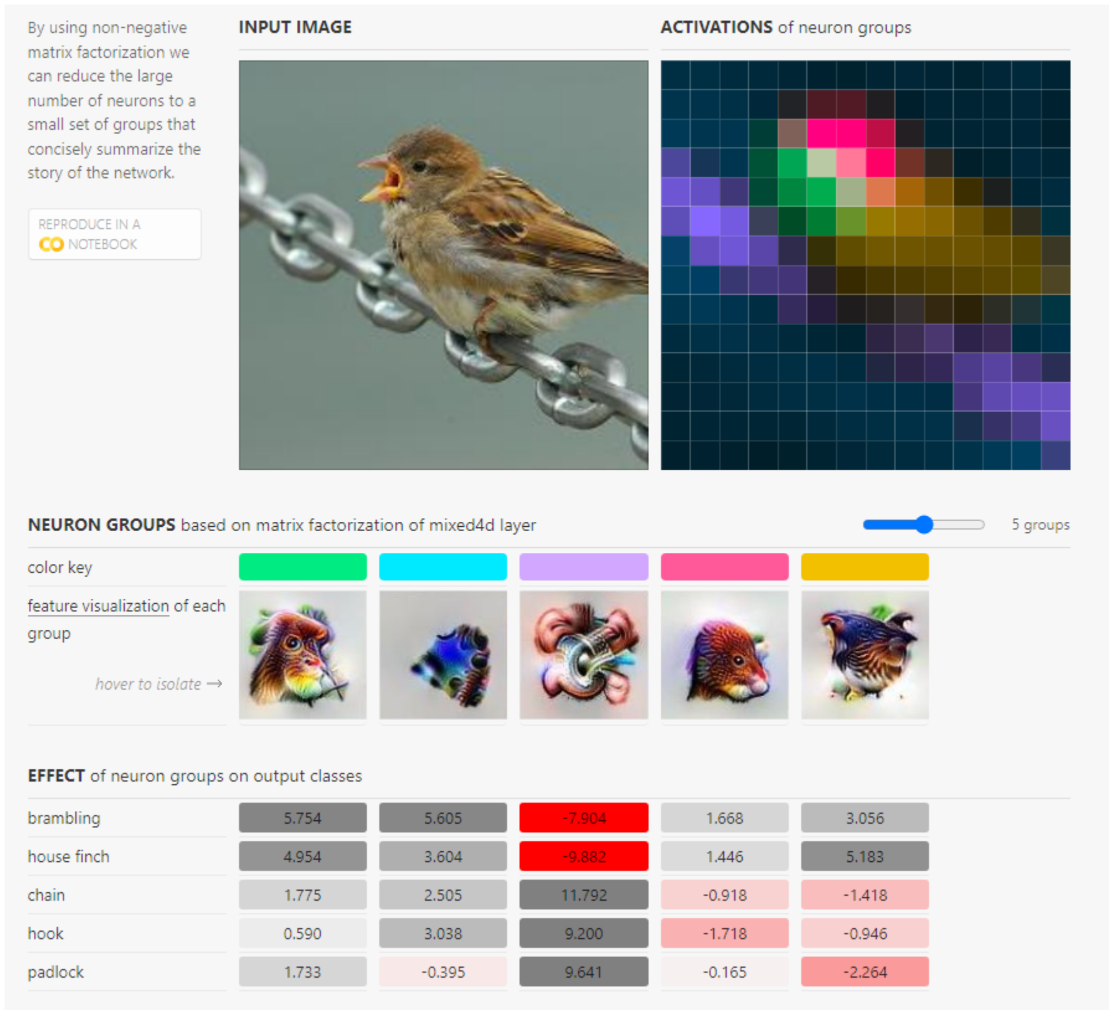
Generators with more complexity stem from reading with more sophistication: compared with a Markov chain, the better performance of Long Short Term Memory neural network architecture [157] can be partially attributed to a more in-depth reading process which takes into account correlations that are more complex than the short horizon a reasonably-sized Markov chain can remember. Attention-based generators, such as GPT [282, 283] and its successors make this even more explicit: the generative power of a Transformer comes from the ability to read the data and only pay attention to what it finds important [364].

A significant focus of recent machine learning research has been on the question of interpretability [260]: making the process of the neural networks more legible to humans (Fig. 7.1). In turn, the more interpretable generators have led to research such as StyleGAN, which is intended to be easier to read for both humans and machines: by using an intermediate latent space, the parameters for navigating that space are less entangled and more directly correspond to interesting vectors of variation, making it easier to read [177]. Ultimately, the development of CLIP [281] and other multi-modal models points to reading being one of the most vital aspects of current machine learning research.

While reading is an intrinsic part of machine learning, it is not confined to neural networks. Procedural generation algorithms like WaveFunctionCollapse [186] also depend on reading. WaveFunctionCollapse performs a generativist reading on the images it uses as input and translates them into a model that can, in turn, be used to generate new examples that imitate structures it has recognized in the input.

7.3 Game Generation

Not every generative system that makes use of reading as part of the generative process necessarily conducts a generativist reading. Gemini [342], a generator of simple abstract games based on Martens et al.'s proceduralization of proceduralist readings [222], conducts proceduralist readings on the games it generates in order to determine whether or not they can be interpreted in a way that matches the arguments



By using non-negative matrix factorization we can reduce the large number of neurons to a small set of groups that concisely summarize the story of the network.

REPRODUCE IN A NOTEBOOK

NEURON GROUPS based on matrix factorization of mixed4d layer — 5 groups

color key	🟩	🟦	🟪	🟥	🟨
feature visualization of each group *hover to isolate →*					

EFFECT of neuron groups on output classes

brambling	5.754	5.605	-7.904	1.668	3.056
house finch	4.954	3.604	-9.882	1.446	5.183
chain	1.775	2.505	11.792	-0.918	-1.418
hook	0.590	3.038	9.200	-1.718	-0.946
padlock	1.733	-0.395	9.641	-0.165	-2.264

Figure 7.1: An example of reading a neural network: summarizing the story of a neural network's interpretation of an image by collecting the activated neurons into groups. From "The Building Blocks of Interpretability" licensed under CC-BY 4.0 [260]

the user has specified they want to make. However, this reading occurs only internally—the things it reads are the incomplete games that it has itself generated—and it does not build a generative model of these games based on its reading, but merely uses its reading to direct its search within the possibility space.

In contrast to Gemini's proceduralist reading approach, the Ludi game generation system reads abstract games in terms of their rules, as formatted in a game description language. Ludi reads game rules through a process of analyzing self-play simulations. Ludi then evaluates their fitness as defined by a set of algorithmically-measured aesthetic criteria [44]. The writing process combines the read game rules into new mixtures of rules. These are added to the collection of game descriptions to create the next generation of games and the reading and writing process repeats. It has been suggested to us that Ludi uses a form of generativist reading: the analysis and evolving process is aimed at generating new game rules that are similar to the game descriptions it read as input, and it builds up a generative model that attempts to describe the possibility space of aesthetically interesting game descriptions.

7.4 Conclusions

As we have seen, reading is a useful analytic lens we can use to better understand how generators process input, including context sensitive generation, chaining generators together into pipelines, and furthering mixed-initiative co-creativity.

A generativist reading is an interpretation of a text into a generator of similar texts. The existing practice of many generative artists and bot creators can be considered as generativist readings, while mechanical generativist readings are foundational to many existing approaches to PCG, including procedural content generation via machine learning. The greater the complexity of the generator, the more sophisticated the reading required: breakdowns in generative pipelines can be caused by a mismatch in complexity between the input and the output, as with Compton et al.'s example of complex input from a Kinect being effectively reduced to button-presses by the mismatch in reading in the pipeline [65].

The distinction between a reading operation and a generativist reading is that a generativist reading outputs a generative system. This is both a reading as a method of analysis (understanding an artifact by constructing a system that could have constructed it) and, in a similar move to Martens et al. on proceduralist readings [222], can be used as a generative operation as one part of a generative system. Anything that has an input and an output can be incorporated into a generative system, including something that inputs or outputs generative systems.

7.5 Acknowledgements

This chapter uses material originally included in “Generators That Read” [200], which I co-wrote with Max Kreminski and Noah Wardrip-Fruin. It was significantly improved based on feedback from anonymous reviewers during the peer-review process, and the parts of the present chapter have been further developed based on subsequent

conversations (particularly comments by Ian Horswill, later conversations with Kate Compton, and my dissertation committee).

Chapter 8

Vivisecting Procgen

What are the lower level components that make up a generator, and how do those components fit together? (RQ2) There is a tendency to write about procgen as a series of algorithms, isolated and independent. However, when procgen is used in practice, it quickly becomes apparent that most generative systems are formed out of complex relationships between many generative operations. While there have been some theoretical frameworks that acknowledge this, particularly the Generative Framework (Sec. 3.7) and in discussions of Game Orchestration [207], this chapter is intended to lay out a general theoretical model for how generative systems are architected. In particular, this modular analysis approach makes it easier to see how individual parts of the system are implemented while also making it easier to discover the poetic implications by comparing how different generative operations are used in similar ways.

Critically, this perspective enables us to move beyond individual artifacts and examine how generative possibility spaces are manipulated. A filter is a higher-order function that subdivides a generative possibility space. Filters are recognizable as branches in the topology of the generative system. Understanding filters allows us to understand how to construct more complex generative systems that intermix additive and subtractive approaches (Sec. 8.1.3).

8.1 Nodes and Edges

We can model a generative system through the lens of *information flow*, thinking of it as specific packets of *data* that exist within the generative possibility space. Following the Generominos [65] approach, consider a generative system as a network of nodes (the processes) connected by edges (the data being passed from one process to the next).

Every edge is always the same data from beginning to end: the data only changes when processed by a node. Nodes have input parameters and output data, and each of these has a specific type. This means that if one node's output data does not match another node's input parameters, if we want to connect them we must transform the data from one type to another by means of a node—in other words, we must perform a generative operation.

The types of data being transmitted and processed can be very simple (including single Boolean values), but they can also be quite complex—up to and including complete generative systems. Likewise, the nodes can range from very simple arithmetic up to complete encapsulated generative systems.

Thus far, this is very similar to the Generative Framework by Compton and Mateas (Sec. 3.7). The difference is in the classification of nodes: while the catalog of types of transformations is useful, for this model it is more effective for us to consider how each node fits into the topology of the generative system. Where the Generative Framework is concerned about *how*, this model is concerned about *what* and *where*.

Therefore, the types of nodes are **input sources**, **output sinks**, **transformation processes** (which include **generators**, **translators**, and **readers**), **filters**, and **evaluators** (Fig. 8.1).

8.1.1 Sources and Sinks

While it might be useful in some circumstances to model the processes outside of the generative system,[1] for the present discussion in this section we can treat them as opaque.

The first external connection node is an information *input source*. The input cards in Generominoes would fall into this category, but it is important to emphasize that *source* is broader than *input* might imply. A source might be outside input, a random seed, or any other form of information that enters the generator in a form that we can process. The important criterion is that it is injecting information into the generative system.

At the other end of the system is the information *output sink*: some kind of process that exports an artifact. Technically, we can just grab an artifact directly from

[1]Especially for external processes that feed back into the system, such as mixed-initiative generators (Sec. 8.4).

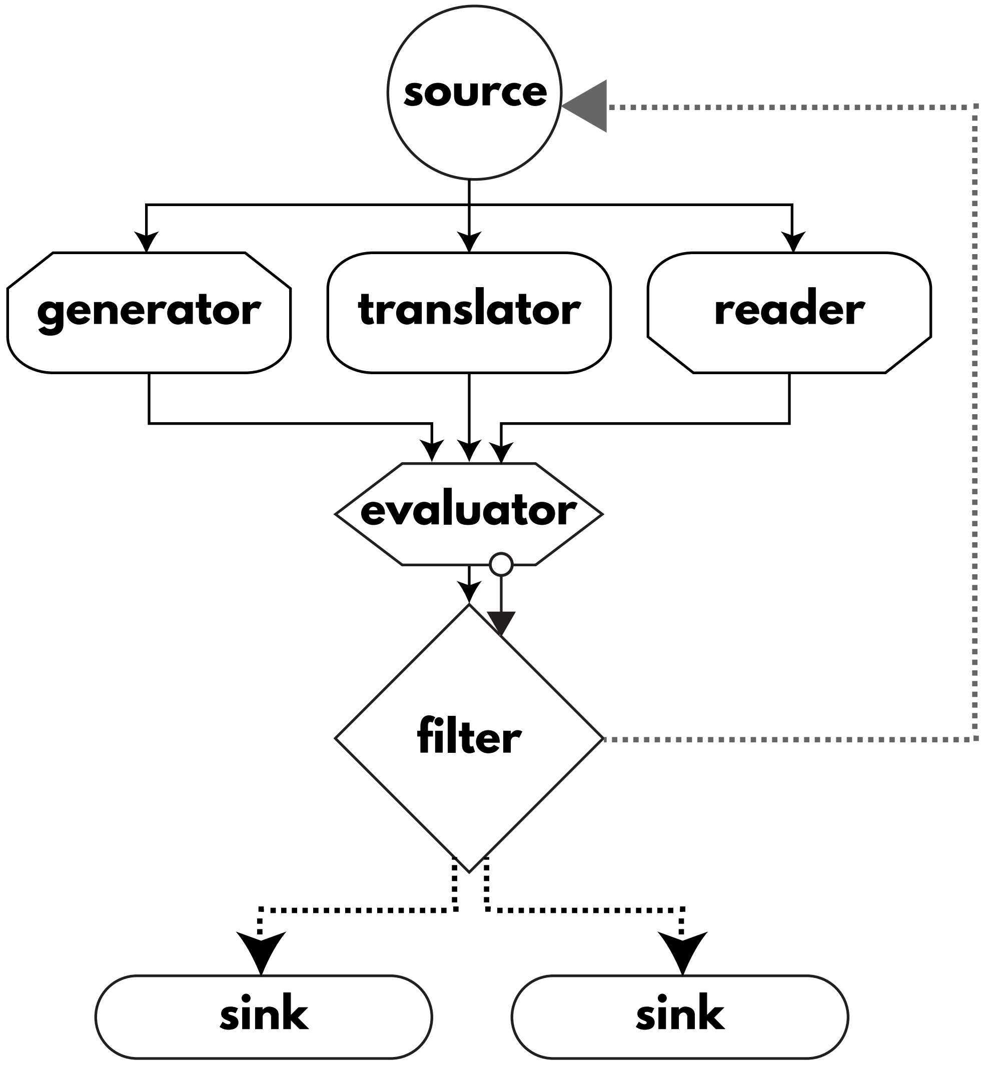


Figure 8.1: Diagram of the different elements of a generative system: parameters are input from *sources* and artifacts are output from *sinks*; in between they are processed by *transformation processes*, ranging from *generators* (expanding data), *translations* (changing data), and *readers* (interpreting data). *Evaluators* annotate the data with explicit or implicit metadata, and *filters* use the annotation metadata to act on the generative possibility space, shaping the new latent space by backpropagation of changes to the parameters or by forward branching.

the output of any node, piping it directly into the sink, though usually there's some final transformation involved to get an artifact in a particular format.

From the point of view of the generative system, an information sink *consumes* information and sends it outside. The system is, admittedly, not likely to notice that it is gone: there's no shortage of non-existent information, after all. Nevertheless, a sink is a node with an input but no (in-system) output. Note that, contrary to the Generative Framework, in this model a search-based technique is *not* a consumer.

We should be careful to note that, while an input source is a source of *varying* information (i.e., parameters), information can also enter the system by being embedded in the processes and topology of the generative system's transformations.

8.1.2 Transformation Processes

Sources add information, sinks consume information. *Transformation processes* change information.

Transformations encompass most of the characteristic parts of the generator. Compton subdivides transformations into content selection, tiles, grammars, parametric, geometric transformations, distributions, agent-based, and machine-learned statistical models [64]. But for the moment I want to concentrate on the topological relationships and look at what they do with information. We can categorize these processes on multiple spectra: **generator processes**, **transformation processes**, and **reading processes**.

Overall, transformation processes are frequently about projecting data from one configuration to another, resulting in the same data type but in a different coordinate system.

A *generator process* is a transformation that outputs information of a different type than was input. This information is technically not being created, but is rather emergent from whatever internal structure the generator has. The formula for the Mandelbrot set generates many surprisingly complex patterns, but all of that supposedly new information comes from the implications hidden in its equation. Someone had to discover the properties of $z_{n+1} = z_n^2 + c$ before we could take advantage of it for our fractals.

Which is an important point: all of the information in a generative system has to come from *somewhere*. While it often looks like a generator transformation is creating information, it is merely revealing some part of the emergent effects of its own internal rules. The structure of a generative system is itself a form of information being input into that system.[2]

Many of the foundational generator processes simply add structure and context to the otherwise meaningless noise, or takes an ordered, linear sequence and chaotically perturbs it (compare Sec. 5.3.2). Generator processes (and generative systems as a whole) are about combining one kind of ordered structure with another kind of chaotic disorder to create emergent effects.

A *translation process* has input and output on the same order (either the exact same data type, or at least not mixing chaos and order). Some simple translation pro-

[2]It follows that the most efficient way to expand the expressive range of a generative system is often to build a second generative system that uses a different structure for similar ends.

cesses that are used in many generators include inverting, threshold functions (including step functions and ReLU), high-pass filters, fast Fourier transforms [77], etc.

A *reading process*, on the other hand, is a much more indirect relationship: it performs a generativist reading on its input (Sec. 7). This often results in an output data type that is very different from the input data type. It can also be quite complex: an AI using regions and pathfinding to interpret a map; computer vision object recognition; interpreting a natural language text.

Care must be taken that the input still matters by the time it reaches the output: because a generativist reading is complex, the information in the process itself can dominate the information from the input, resulting in the original information being lost to the noise.[3]

The type of transformation, then, is defined by the shape of its inputs and outputs: small input, large output is a generator; big input, small output is a reader. In each case, the transformation doesn't know or care about who is consuming the data after it is output: it doesn't have a way to sense anything about what happens to the output unless some other process feeds information about the output back in as input parameters. Some implementations of generative system building tools *do* care, though often this is mostly because it can be more efficient to work out which generative operations to run by starting from the output sink and working backwards.

If a generative operation *does* get information about its output, it requires a higher-order structure in the topology of the generator. Thus, the information can only

[3]Compton's *Generominos* is heavily concerned with this question of preserving data relations [65], which makes it a natural complement to this framework.

reach it by passing through a filter operation.

8.1.3 Filters and Evaluators

Transformations are not the only kind of feature in a generative system. Transformations act on the *data*, but there is another class of generative operations that act on the *generative possibility space*.

Rather than transforming information, a *filter* changes the shape of the generative possibility space by pruning unwanted parts of it.

We can recognize a filter in a generative system by its topology in the network of nodes. A correct-by-construction generator doesn't need a filter: therefore, there are no cycles in its graph. A filter introduces either a cycle or a branch: because some parts of the generative possibility space have been rejected, that information must be back-propagated into earlier parts of the system; or the path of execution can divide into two possibility spaces.[4] A generate-and-test loop is one of the simplest forms of filters, but other filters include search-based generators, PCGRL, and PCGML (Fig.3.1). Instead of propagating backwards, a filter can also branch the execution path going forward: each separate branch gets a different subset of the generative possibility space.

The signature topological pattern is that the output of the operation branches. Many transformation operations have multiple subscribers for the data they output, but that doesn't constitute an actual OR branch. A filter, in contrast, has multiple outputs (or, equivalently, sometimes fails to output for some inputs).

[4]Or the generator can fail and produce nothing, which is technically possible but usually undesirable.

A filter requires an *evaluator*, some function that defines the criteria for shaping the generative possibility space.

The closest equivalent in the Compton and Mateas Generative Framework is the category of consumers of generative pipelines [64]. The Generative Framework talks about methods that consume pipelines. While this perspective makes a certain amount of sense, I find it more useful to conceptualize it as operating on the generative possibility space of the generator's artifacts—the sum total of possible artifacts—rather than speaking of them as consuming the pipeline itself. Additionally, in contrast to the Generative Framework, the present model makes the identifying mark of a filter *topological*: the cycle in the graph. *Consuming a generative pipeline* is the same as *altering the generative possibility space*, but discussing it in terms of the generative possibility space makes it clearer what is being consumed and altered. The topological view of filtering makes it much more tractable to for analysis.

Viewing filters through the lens of the generative system's topology gives us a clearer view of the implications of our architectural decisions. The backpropagation is required because a deterministic system needs additional information to avoid producing the exact same outcome. This can be a minimal amount of information: the simple signal that *this was rejected* can be answered by running it again with a different seed. More complex filters are possible when we start sending more information back: Search-based PCG [356], reinforcement learning PCG (RLPCG) [193], machine learning PCG (PCGML) [344] and so on are filters, and we can effectively model them as part of the topology of a generator, rather than as monolithic types.

8.1.4 Correct by Construction

The generative system topology that appears to be the simplest is a generator that is correct by construction. In the procgen space this has taken on a slightly different meaning than in general computer science. The original use of the term was related to formal proofs, in the sense of Dijkstra's structured programming [81, 88]. Generally speaking, we don't do formal proofs of generative systems.[5]

In procgen, "correct by construction" is often closer to the sense in constraint solving: rather than our program, our *solution* is what is correct.[6] A subtle difference, but significant: we're not trying to prove the correctness of our *generator* per se, we're trying to ensure that the *thing it generates* always meets specifications, by engineering the generative system so that any artifact it outputs meets our specification by definition.

So a correct-by-construction generator looks straightforward: often a linear chain of operations. But its topology hides significant information: the operations had to be arranged in such a way that only valid results are possible. The level generator in *Spelunky* is a case in point: a very simple grammar on the surface, but the rules only allow levels that have a valid path from entrance to exit [385]. Many correct-by-construction generative systems rely on this kind of hidden and implicit knowledge encoding to ensure that the artifacts they generate can never fail to match the specifi-

[5]Given the aesthetic purposes of most generative systems, the problems we try to solve are usually under-specified: we're often not sure if what we think we want is what we actually need, even before we get to the cursed [169] and wicked [292] problems. Still, it is useful to know that a given generative system is doing what we think it is doing.

[6]Such as in this discussion by Gaël et al. on a director AI that can vary player access to subsets of a level [129].

cations.

A correct-by-construction generator doesn't include an explicit filtering operation to trim down the generative possibility space. Instead, it has been carefully designed so that it won't go outside of the valid possibilities to begin with.

8.2 Encapsulation

A generative operation can contain an entire generative system. In this case, we can regard the entire encapsulated generative system as performing the role of one of the types of generative operations—often a generator or a filter, but potentially any of the types. This encapsulation makes it simple to consider each part of the generative system in isolation, making analysis of large systems much more tractable.

It also means that we can side-step any questions about what makes an atomic generative operation. Any generative operation can be regarded as an atomic unit if its contribution to altering the artifacts in the system can be reduced to its input parameters and output results. We can consider it in isolation and only concern ourselves with its subatomic components if the inner operation of that particular subsystem is of interest.

The most obvious example of this is when the encapsulated operation can be represented by a side-effect-free deterministic function $y = f(x)$. However, it can also be useful to think of transformations as mathematical relations (given x, produce some y, such that $p(x, y)$ is true). From an encapsulation perspective, we can think of such a system as a single function $f'(x) = \text{generate\_and\_test}(\text{generate\_y}, p)$. A generative system does not necessarily need to be implemented as a linear sequence of operations in sequential time (Sec. 8.7).

This is most obvious with constraint satisfaction. A constraint solver, in isolation, takes a set of rules as parameters and outputs a result that satisfies the constraints described by those rules. This is relatively straightforward a transformation operation, though it can be deployed as anything from a generator process (simple rules with complex results) or a reading process (complex rules but simple results).

When we look inside the constraint solver, we see where this versatility comes from: the rules both define a design space and constrain it by pruning away the undesired parts of that design space. Therefore, the process of using a constraint solver involves both additive and subtractive elements, as described by A. M. Smith [318, p.127].

A generate-and-test generative pipeline architecture operates by first building up a generative space with additive operations. Once the generative space is defined, filters are used to subtract down to the final desired generative space. With constraint satisfaction, the additive and subtractive moves can be interwoven. In fact, within the constraint solver, the order of operations is decided by the constraint solver. Usually, the designer can write the rules without caring about the order of operations, and most constraint solver specification languages are declarative rather than imperative.

When considered as a black box, the constraint solver can, of course, be represented by its inputs and outputs like the other generative operations. Depending on the configuration, we can regard it as a transformation or a filter. However, from the perspective of the designer, the process of designing the constraint rules makes the design process a matter of refining the design space model. This is evident when we consider that, unlike other filters, a constraint solver might have, as its output, a generative space larger than its input, if the constraint rules define a generative possibility space that exhibits emergence. The new information is from solving the (potentially NP-hard) constraint problem.

Generative systems are not monoliths. Likewise, most generative operations are not atomic. Just as knowing about the modular parts of a generative system is often necessary to understand it, understanding when a generative operation is an encapsulated generative system can inform our understanding of its operation.

8.3 Latent Space and Expressive Range

The *expressive range* of a generative system is the "*potential* range of content the system might be able to create" [325]. A *generative space* is the possible output of a specific configuration of the generative system and its input parameters [325].

Latent space is the corresponding concept from machine learning, where the embedding of complex data is compressed into a "reduced-dimensionality vector space" [211]. A latent space—in the narrow, formal sense—is therefore a vector representation of the state of a generative space at a particular point in the generator. Both the procgen field and machine learning have tools for mapping and exploring the unrealized latent space, including expressive range analysis [308,326,341], latent space cartography [211], dimensionality reduction [362], and feature visualization [259].

8.4 Mixed-Initiative Co-creation and Interactivity

These models of a generative system don't need to be self-contained. Indeed, any generative system that displays a result and allows for user control over the parameters can be used to construct a mixed initiative system (Fig. 8.2). From the perspective of the system, we can treat the human as a kind of external filter. Often this incorporation of human input is omitted from our consideration of how generative systems operate.

Usually, what we think of as mixed-initiative generative systems are the generative systems that explicitly incorporate a user into the generative system (Fig. 8.3). The interactivity happens withing a defined subset of the generative system, often acting as an evaluator for a filter.

The user's input can technically be fed into any of the generative operations, or interpreted through a reading process first, or any number of other configurations. However, they all revolve around the human acting as a part of the filtering process, because the basic topology is that the display of an artifact is (directly or indirectly) causing a change in the system. The change is usually unpredictable, and sometimes acts on the meta-level, but it always incorporates the loop.[7]

Understanding the grokloop[8] that this cycle forms is key to understanding how this interaction affects the design and analysis of our generative system.

[7]An exception could be argued if the user was making the choices without ever seeing an artifact, but I cannot think of any practical uses for that pattern. Write to me if you have an example.

[8]Grokloop: A name for "the hypothesis-action-evaluation cycle when it is used to explore a generative space" [59, p.524] See Compton's discussion of grokloops and casual creators [59, p.124].

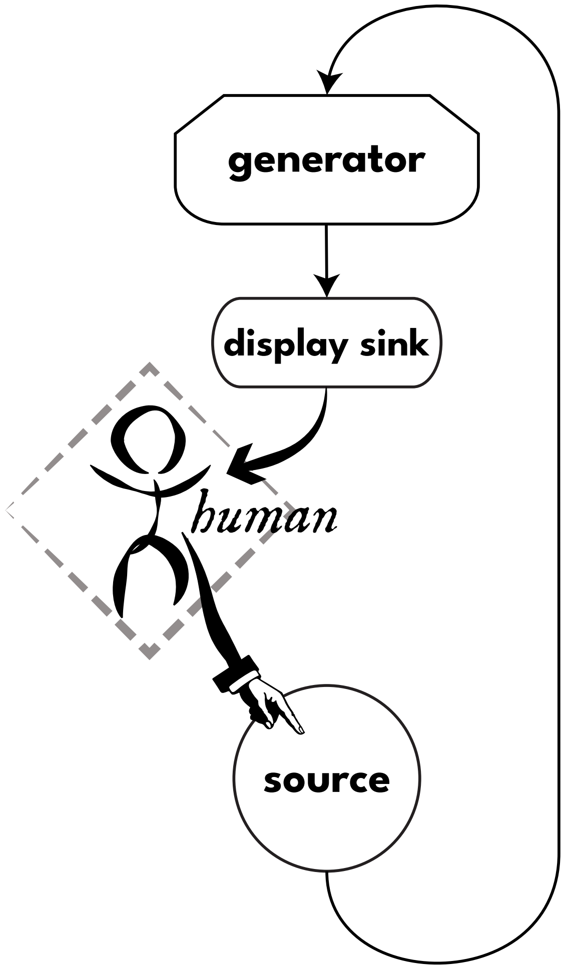


Figure 8.2: The simplest way to incorporate human input is by displaying some output and receiving input that changes the parameters. From the perspective of the system, the human is acting as a filter.

Mixed-Initiative generation has enough depth to sustain multiple dissertations (not least of which is [59]) so for our immediate purposes it is enough to note that this constitutes a special case of the filter/evaluator pattern.

8.5 Generating a Generator

Filters are higher-order functions that subdivide the generative possibility space. I want to briefly mention a further level of generator abstraction: generating a generative system. A generative pipeline can be represented as a graph, graphs can be generated, and it is a small but significant step to then execute that generated pipeline.

Generating generative pipelines is a less common application: the difficulty of constructing a generative system is compounded when you try to construct a generative system with a generative system. However, some domains feature generative pipeline generation as their main form of expression (e.g. livecoding, see Sec. 4.2.2.1) and enough examples of generative system generators exist (e.g. [191]) that I think it is clear that having a better theory of what a generative system *is* will make the problem much more tractable.

The possibility of generating generative systems is another reason to escape the monolithic model: it is much easier to *start* asking the question when we can think of different generators as being constructed from DNA rather than as unrelated species. Additionally, we aren't limited to generating pipelines with pipelines, but that requires

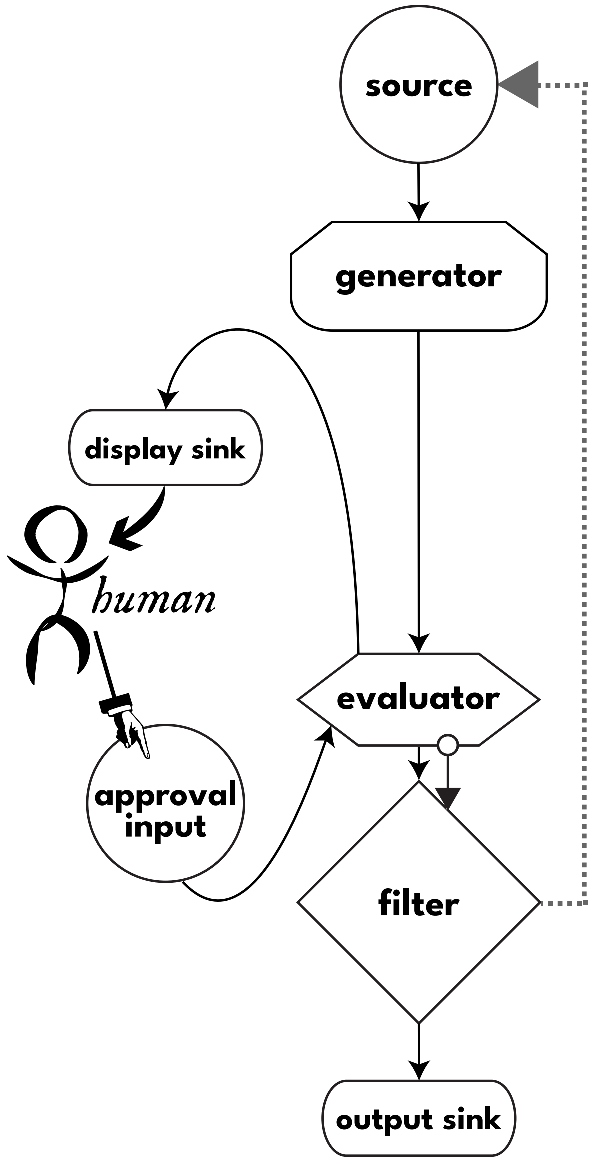


Figure 8.3: One way a human can be incorporated into the generator is as part of the evaluator. Here, the human is acting as the critic (3.6) or evaluation function (8.1.3) annotating the artifacts.

rethinking the topology of the generator from a more ecological perspective (Sec. 12.4.2).

8.6 Aboutness and the Labeling Problem

It is important to note that examining the generative operations and topology in isolation will not tell you what the generative system is going to be used for. You can draw some distinctions around what domain the artifacts are likely or intended to be used for, though that is just an affinity rather than what the generator is about. There is no about. It is precisely because context and framing matter that labeling is important.

As a field, procgen is prone to falling prey to the fallacy that I've termed the *labeling problem* and Kate Compton has referred to as "aboutness." As in, a generative system *does not have an aboutness*, and the label we apply to it can equally obscure what it can be used for.

An example from the related field of audio production: Digital Audio Workstations (DAW) are complex pieces of software, with instruments that often have presets for particular effects. While some presets are labeled with straightforward descriptions of the sound they are emulating, others are labeled fancifully or abstractly. This extends far beyond music: the limitations of our under-specified common vocabulary for sounds are compounded by the fact that sound effects often originate from objects very different than what they eventually represent on-screen [376]. While you could limit yourself to the use implied by the label on the preset, creative off-label use is key to mastering sound effects and music production.

In procgen, unfortunately, we tend to confine ourselves to the label more often than not. Take the use-case of generating a mountain: being a traditionalist, you might first reach for the Perlin noise. But that comes bundled with a lot of assumptions: that what you need is a heightmap, that you want to generate the terrain first and the things on the terrain after, and even that you want the mountain to have height at all. But there are examples that break all of these assumptions: for example, if our level designers made ski-trails for the next SSX game, and now you need to make a mountain that fits them [162]. Or perhaps our gameplay requires flat areas, and we need to generate a level that feels like it is in the mountains without using elevation (*Diablo*[9] and *Moon Hunters*, to name two examples). As Kate Compton has argued, there is no thing-generator, "There is only a thing-aboutness generator" [61].[10]

To summarize: stop thinking of L-systems as the first thing to use when you think about generating trees, but also use L-systems for more than just trees.

8.6.1 Framing and Information

Information exists inside the generator, embedded in the generative operations and in the topology of the generative system. However, aboutness demonstrates that it

[9]Thanks to Kate Compton for reminding me of this.

[10]In parallel, Kate Compton has informally proposed a model of procgen with four orthogonal considerations [60]: [11]. We can label them as:

1. *Subjects*, or the things that we are using procgen to build.
2. *Methods*, or the tools that we use to generate things.
3. *Problems*, or the failures and limitations of a given generator or approach
4. *Interfaces*, or the tools for exploring the generative space.

also exists outside the generator, in the *framing*.

The meaning of a generated artifact exists in the user's head: while the artifact might exist independently of perception, the perception dictates our understanding. The interpretation of the artifact depends on the context the user understands it in (Sec. 5.6.3.3).

This is why framing is such a critical part of a generative system. *Framing* refers to "information supplied alongside a creative work," [75] and gives it context. This is a generalization of the concept of framing in computational creativity theory [56].

8.7 Time

One thing that this framework does not explicitly model is the role of time. It approaches generative systems that produce discrete artifacts. This stateless, pure-function approach makes the analysis of very complex systems tractable, with different elements being able to be considered in isolation. However, other ways of dealing with time are possible, particularly in terms of the order of operations within a generator, or how the output of a generator changes over time.

Generative operations do not need to be executed linearly. While there are dependencies, often there are many operations that can be run in parallel. We can even take advantage of the mathematical relationship between certain operations to have them share implicit data without needing explicit parameters. An example of this is that operations branching off from a shared data source that affects their parameterization—

such as signed distance fields (SDFs) or Worley Noise [380]—can be layered on top of each other without needing direct communication. A simple application of this is combining several operations to texture different parts of a cobblestone texture based on Worley noise: the textures for the top, edges, and gaps between the stones can be calculated independently, without needing further calculations.

Further, instead of producing discrete artifacts, some generative systems are better modeled as producing *streams*. This is most obvious with generators that run autonomously, producing an infinite stream of output, such as an infinite live stream of generated technical death metal music [52]. An entire musical performance could be considered to be the finished artifact, and analyzed from that perspective. However, an unending generator or a generator that can be altered while it is running can sometimes be better understood by taking time into account.

Music is an obvious example of time representation: many music generators have a way to represent time, sometimes even feeding the output back into the system as input. A less obvious example is animation, where an animation performance changes over time in response to outside stimuli. Livecoding—being a performance that often includes both audio and visually animated elements being controlled by a programmer modifying the code as it is running—naturally incorporates a generative system that changes over time.

The general case is that any generative system that can have its parameters or topology modified while it is in operation includes a notion of time. Modifying parameters is relatively straightforward, but there are several ways to represent it. One way is for each beat or tick of the generative system could be regarded as a separate run of the generator, with the artifacts being the individual notes or beats. Another way is to model it in a stateless, timeless way, with time as just another dimension of the artifact.

Individual notes or moments is useful way to model a stream with an indeterminate end, similar to Minecraft breaking everything down into regions of blocks. Modeling an artifact with a time embedding as a positional embeddingis used with transformers [364], particularly relative positional embedding [309], and can be extended to other generative systems.

Modifying the generative topology can get very complex very quickly. Generative systems that produce streams of generative systems have a further dimension of complication: this can happen in livecoding, for example, where the running system ends up in a state that isn't described by any explicit representation of the system [319].

With any generative system that will be run over time, it is worth thinking about how its output will relate to changes over time. In a Minecraft [238] world, for example, the generator is only run when a player gets close enough to the edge of the world to reveal a new region. Because players sometimes carry worlds from version to version, a map can end up with different regions generated by different versions of the software, sometimes resulting in anomalies that couldn't have been produced by either of the versions on their own.

8.8 Conclusion

Generative systems don't create information out of nothing: all of the information that comes out of it had to first be put into it. There are many ways to leverage the generative operations to do this more effectively, with less effort, but ultimately all generative systems must get their information from somewhere, incorporate it in some way, and put it to use at the right time and with the right context.

Understanding the topology of a generative system is a critical part of our analysis. A lot of the information in a generator is embedded in the topology. However, it must work in concert with other forms of interpretation and evaluation, such as poetics (Chap. 5) and generativist reading (Chap. 6).

Chapter 9

Constraint Solving in the Wild with WaveFunctionCollapse

Having a general framework of generative systems is a start, but looking at specific generative systems is also important for answering our question about specific components. (RQ2) In this chapter, I describe one family of generative systems, popularly labeled as WaveFunctionCollapse, or WFC.

Specifics are important for research. One way to contribute to procgen research is by demonstrating either how to reconstruct a technique or how a technique can be applied to different ends. By interpreting generators as they exist in the wild, we can draw connections that go beyond a particular case study. As we document specific systems, our ability to generalize increases and our models are better able to approximate what is possible. In this chapter, I examine a specific technique, or category of techniques, and demonstrate how it connects to past research and current possibilities.

At one point, academic discussion of procedural generation was roughly divided into constructive generation and search-based techniques (cf. 3.2.1). Other ways of building a generative system had been discussed by multiple academics, such as constraint solving and machine learning for procedural content generation [321, 344], but they weren't widely implemented in the videogame industry. In the fall of 2016 a new approach to procedural content generation burst onto the scene. It was a surprising example of both a constraint satisfaction problem (CSP) and machine learning (ML) techniques: WaveFunctionCollapse (often known simply as WFC).

WFC was developed by Maxim Gumin and the `C#` code was released on GitHub in 2016; six years later the repository also links to several dozen different ports, libraries, and implementations.[1] WFC uses constraint solving and machine learning to translate minuscule, artist-created training images into large, expressive generated content. It has been used on everything from small-scale game jam and PICO-8 fantasy console projects to larger commercial games released on the Nintendo Switch,[2] and has since been taught to undergraduates at several universities and used in technical games research.

This chapter is about the emergence of WFC, building on the examination started in 2017 with a workshop paper [186], and paralleling a journal article in 2022 [188]. I present a compact overview of the family of algorithms that make up WFC, including the terminology and concepts that it uses or introduces, such as tiles, patterns, propagation, and the use of inputs and outputs in the algorithm's pipeline.

[1] `https://github.com/mxgmn/WaveFunctionCollapse`

[2] Respectively: `https://arcadia-clojure.itch.io/proc-skater-2016`, `https://trasevol.dog/2017/09/01/di19/`, and *Bad North* (2018)

This chapter includes an approachable high-level pseudo-code walkthrough of the operation of WFC, towards the aim of interpreting generators in the wild for an academic audience and making the algorithm more approachable for practitioners. I also describe my rational reconstruction of WFC, which I use as a testbed for experiments on the contributions of the different elements in its pipeline. As part of demonstrating how researching a particular algorithm can further the field as a whole, I also include a discussion of how the wider world of research into constraint solving and machine learning (including deep machine learning) can factor into the algorithm. Additionally, I trace the adoption of WFC across academic, industrial, and artistic users, demonstrating how progress in procgen research spreads.

9.1 Research Context

WFC can be positioned as a development of a number of lines of previous research. While this does not lessen the surprise of its viral adoption, it does provide context.

9.1.1 Texture Synthesis

In computer graphics, *texture synthesis* is the problem of generating a large (and often seamlessly tiling) output image with texture resembling that of a smaller input image [99]. In many texture synthesis approaches (e.g. the work of Liang et al. [204]), the input and output images are characterized in terms of the local patterns they contain, where these patterns are typically sub-images of just a few pixels in width (e.g. 5-by-5 pixel windows). Many texture synthesis algorithms explicitly intend to produce outputs such that every local pattern in the output resembles a local pattern in the input. In the visual setting of graphics, this resemblance need not be exact pixel-for-pixel matching and is often judged based on a distance metric (e.g. Euclidean distance of pixel color vectors) that judges some colors to be closer than others. By contrast, exact matching is the only sense of resemblance present in WFC.

In Liang's method [204], the output image is grown incrementally. Part-way through the generation process, a large region of the output has already been generated, but some remains unresolved. A location on the border of this region is selected, and the surrounding already-chosen pixels (the context) are used to query an index of patterns generated from the source image. A pattern with similar local pixels is retrieved and used to paint in a few more pixels of the output image, growing the region of completed pixels. WFC also grows its output image incrementally, expanding the known regions of the output by completing them with details from local patterns of the input image. However, it also considers the not-yet-known space because of its consideration for exact pattern matching.

While WFC is loosely inspired by quantum mechanics,[3] Gumin was also inspired by Paul Merrell's *discrete* synthesis approach. Merrell, in turn, was inspired by texture synthesis, but he focused on the problem of generating 3D geometric models [230]. In this setting, we want to generate a new (typically large) 3D model which

[3]Very loosely, and mostly confined to how it models the superposition of possible image states. As Gumin explains, "The coefficients in these superpositions are real numbers, not complex numbers, so it doesn't do the actual quantum mechanics, but it was inspired by QM." [145]

is made up of components and arrangements taken from a (typically small) 3D model provided by a human artist. Per texture synthesis traditions, artifacts are characterized in terms of their local patterns on a regular grid. Instead of blendable pixel colors, however, Merrell's discrete model synthesis aims to exactly reuse rigid geometric chunks (as it is far from obvious how to simply blend geometric chunks). Merrell's key contribution is to pose the output sampling problem as one of constraint solving: two pieces of a 3D model can only be used together in the output if they were used together somewhere in the input.

As another influence, in personal correspondence with us Gumin described how he was influenced by convolutional neural network style transfer but found it lacking for videogame level generation. He experimented with several approaches to model and texture generation, looking for a texture synthesis algorithm with strong local similarity, where each $N \times N$ pattern in the output could be traced to a specific input pattern. Gumin's intent was to capture the rules for how the source image was made.

Gumin's SynTex project [142] implemented several texture synthesis methods, yielding attractive results for game texture images but nonsensical outputs for non-texture images where pixel-grid analysis destroyed the visual semantics of structured objects (e.g. swords). In his ConvChain project [141], he experimented with an approach based on Markov Chain Monte Carlo (MCMC), a statistical sampling approach that directly measures how likely an output image is under the distribution of local patterns implied by the input image. Statistical modeling is also present, if much less explicitly, in the notion of entropy used in Gumin's later WFC algorithm.

9.1.2 Constraint Solving

Despite some historic overlap (e.g. computer vision) the field of artificial intelligence (AI), has been somewhat disconnected from computer graphics until relatively recently. Recent innovations in style transfer were sparked by a breakthrough in using deep convolutional neural network classifiers to mimic artistic styles [122]. This has led to a flood of related research, along with the exploration of other applications of neural networks to graphics. But AI includes more than just neural networks.

In the field of artificial intelligence, constraint solving uses ideas from knowledge representation and search to model both continuous and combinatorial search-and-optimization problems and solve them with domain-independent algorithms [299, Chap. 6].

Constraint satisfaction problems (CSPs) are typically defined in terms of decision variables and values. In the context of WFC-style image generation, there is a variable associated with each location in the output image. In a solution to the problem (called an assignment), each variable takes on a value. Depending on the context, values may come from continuous or discrete domains. For the task addressed by WFC, the values are associated with the discrete set of unique local patterns in the input image. The choice to assign a specific variable a specific value will often constrain the available choices that can be made for other variables. Constraints relate the legal combination of values that a set of variables might take on in a valid assignment. For the image generation task, we want to model the idea that the patterns chosen at each location in the output are compatible in terms of exact matches for the pixels in which their associated local windows overlap.

The goal of an algorithm for solving CSPs (a solver) is to find a total assignment (an assignment for every variable) such that no constraints are violated. Although there are many different approaches to constraint solving, most operate by searching in the space of partial assignments. That is, they search the space of incomplete solutions, not generating a single candidate solution until that solution is known to be free of conflicts (constraint violations). The solver repeatedly *selects* an unassigned variable and then *decides* on a value to assign from the variable's domain. If the solver encounters a partial assignment for which no subsequent variables can be assigned without violating constraints, the solver typically backtracks on a recent decision—backing out of a dead-end by moving to another point in the space of partial assignments.

To the skeleton of backtracking search sketched above, advanced constraint solving methods add improvements that attempt to speed up identification of a legal total assignment. Some *heuristics* (either domain-specific or domain-independent) aid the selection of a promising variable to select next, while others aid the decision of a promising value to assign for that variable. The addition of heuristics typically alter the order in which the solver explores the space without impacting completeness guarantees (i.e., that the solver will return a solution in bounded time if at least one exists).

Complementary to heuristics, *constraint propagation* methods do additional bookkeeping in order to prune away values from domains that would lead to dead-ends later. Constraint propagation ideally allows a solver to skip past fruitless search without impacting the order in which the space is explored. AC-3 is a well-known constraint propagation algorithm [299, Chap. 6]. Although AC-3 and other propagators can end up making assignments to variables as part of their operation, they are not complete solvers by themselves. Propagators are typically run after each choice by a solver in order to simplify the remaining search problem. For a game-focused audience, I refer the reader to the *Game AI Pro 2* book chapter, “Rolling Your Own Finite-Domain Constraint Solver” [108] for more details.

9.1.3 Constraint Solving in PCG

Although there are a few examples of note, until recently constraint solving was mostly overlooked for the purposes of content generation. Taxonomies of PCG, such as in the notable search-based PCG survey [356] (cf. Sec. 3.2.1) do not account for approaches to content generation that are neither directly constructive nor perform their search at the level of completed candidate designs. The concept of working with partial designs is part of what makes the animations derived from WFC executions so visually stunning—we are not used to seeing our generators work this way.

Constraint-based PCG methods are often associated with making strong guarantees about outputs, as well as having the cost of those guarantees paid in unpredictability of total running time. Most backtracking solvers yield good performance on their associated search tasks for real world problems, but this outcome is hard to characterize in terms of theory (where exponential worst case analyses are uninformative). Horswill and Foged [161] describe a “fast” method for populating a level design with content under strong playability guarantees. Their algorithm is based on backtracking search with (AC-3) constraint propagation. Although it makes only modest demands on processor and memory resources, it is expected to be used by programmers who are at least moderately literate in search algorithm design.

In G. Smith's Tanagra system [328], a mixed-initiative platformer level design tool, the Choco [277] solver is invoked to solve a specific geometric layout subproblem in the overall level design process. In this system, the user is in a designer role rather than a programmer role. When the solver determines that the given CSP is impossible to solve (we say the constraints are unsatisfiable), it signals to the larger tool that other decisions about the working level design, such as what activity the player performs on each platform, need to be relaxed (backtracked). Although Tanagra illustrates that CSPs need not only be created by programmers (they can be assembled programmatically from the data input into a graphical user interface), backtracking still plays a major role. In surprising contrast, Gumin's original formulation of WaveFunctionCollapse does not make use of backtrack.

9.1.4 ASP in PCG

Answer set programming (ASP) is a form of logic programming targeted at modeling combinatorial search and optimization problems [125]. In ASP, low-level constraints are automatically derived from the high-level rules in a problem formulation program, and the implied CSP is solved using algorithms rooted in the SAT/SMT literature [126].

A. M. Smith proposed the use of ASP in PCG [321] within the paradigm of modeling design spaces. Rather than directly aiming to code an algorithm for generating content, this approach suggests we should declaratively model the space of content we want to see and let a domain-independent solver take care of the procedural aspects. Although programmers using ASP need not have or use any knowledge of search algorithm design, they are expected to be familiar with the declarative programming paradigm and Prolog-like syntax. This background is not common amongst technical artists who were recently excited to find WaveFunctionCollapse.

Modern answer set solvers (such as Clingo [175]) allow for specification of custom heuristics, externally checked constraints interleaved with the search process, and hooks for scripting languages in the service of integrating solvers with outside environments. These custom extension points in Clingo have been used to replicate the incremental growth behavior of WFC to produce an animated[4] log of the solver's decisions.

9.2 Illuminating Gumin's Reference Implementation of WFC

Listing 9.1: WFC Propagation Step.

```
function Propagate(wave_matrix, adjacencies):
    Initialize the propagation stack to contain the last cell observed.
    while there are cells on the propagation stack:
        for neighbor of this cell:
```

[4]"Visualizing WaveFunctionCollapse reimplemented in ASP: 46x46 Flowers N=3, using clingo's default heuristic (VSIDS). Just 3 conflicts." https://twitter.com/rndmcnlly/status/867605489789489152

```
            for neighbor_pattern in neighbor_domain(neighbor):
              if not( adjacency(original_pattern, neighbor_pattern) ):
                Decrement count of neighbor_pattern.
              if count is zero:
                Remove neighbor_pattern from neighbor_cell.
                Add neighbor_cell to the propagation stack.
  return wave_matrix
```

9.2.1 Rational Reconstruction

Now that I have introduced a high-level summary of the reference implementation, we can break it down further, into modular substages. WFC is not a monolithic generator, but rather a multi-stage pipeline of generators and data transformations (Fig. 9.1). Many variations on this pipeline exist in the wild, but all of the members of this family share the core adjacency-learning and constraint solving stages. Breaking the general case down to component steps, the stages in the complete list are: making data input interpretable as tiles; tile classification and grouping; pattern classification and adjacency learning; constraint solving; and rendering the result of the constraint solver into tile data. The description that follows is intended to be detailed enough to guide a reader in producing their own implementation of WFC.[5]

9.2.2 Adjacencies

Conventionally, WFC operates on a rectilinear grid. This gives us an easy way to define the cells and adjacency edges: the cells are each intersection on the grid and

[5]Our Python reconstruction can be found at `https://github.com/ikarth/wfc_2019f`

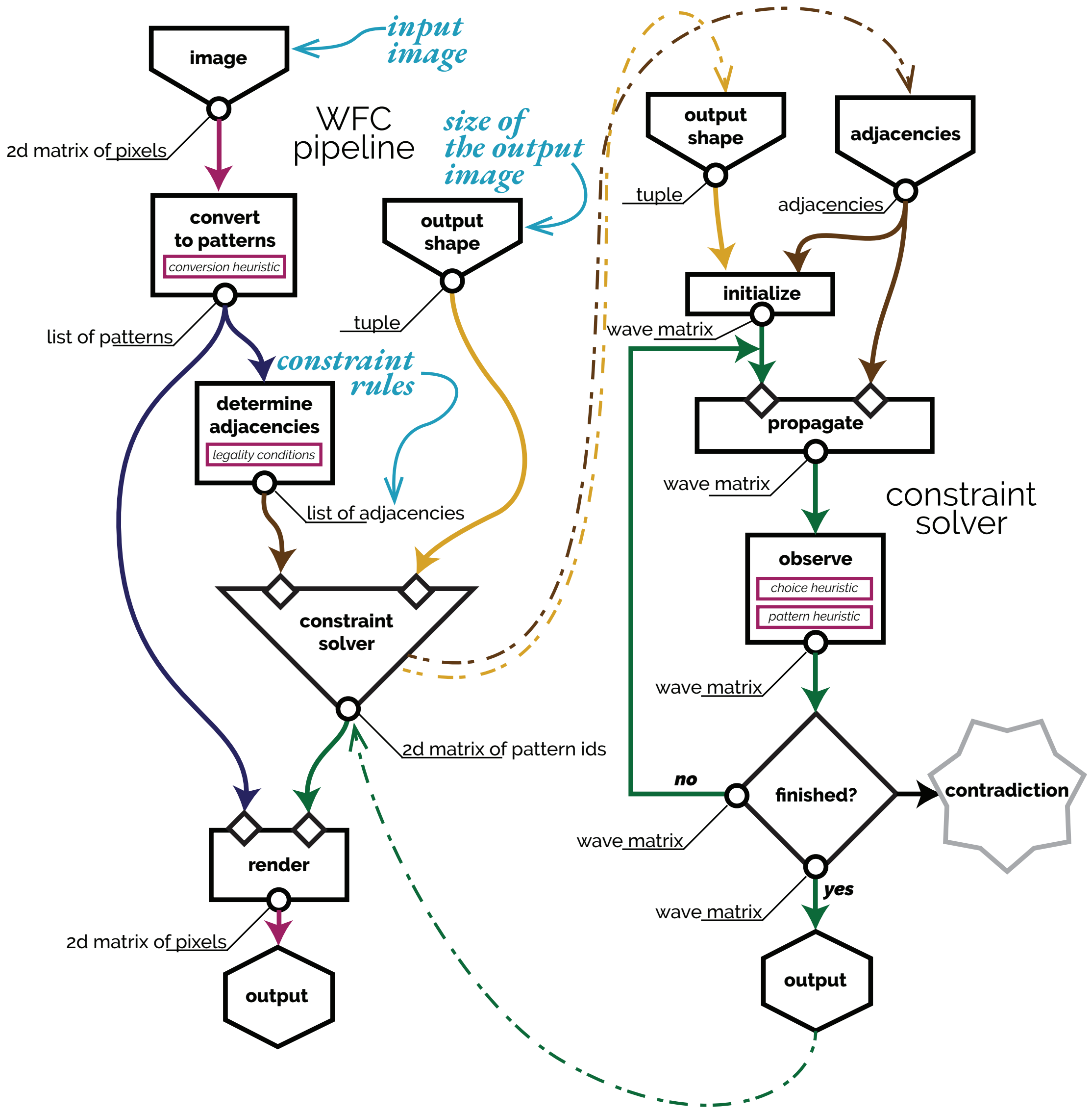


Figure 9.1: Diagram of data and control flow in the WFC generative pipeline. Given an image, WFC finds patterns within it and then determines the valid adjacencies between those patterns. These become the constraint rules for the solver—one of the two main data types it uses. The other major data in the solver is the wave matrix, a Boolean representation of all of the pixels in the image indexed against all of the patterns. The solver alternates between propagating the implications of the constraint rules (propagate) and constraining a domain via assignment (observe).

the edges are the lines between them. Many WFC implementations take advantage of this to optimize performance based on assumptions about the grid geometry.

However, WFC can work on any graph for which a well-defined adjacency function can be specified. For example, Stålberg's experiments with triangular meshes[6] or Florian Drux's GraphWaveFunctionCollapse.[7] These can be non-spatial: to create a looping animation Matt Rix added a time edge.[8] In fact, Gumin notes that d-dimensional WFC can capture the behavior of any $(d-1)$-dimensional cellular automata [145]. Similarly, Martin O'Leary's WFC-based poetry generator[9] uses non-local edges for rhyming patterns and scansion. I call the basic unit which patterns are constructed out of *tiles*. Tiles, in my terminology, are the atomic units of the input data: the palette of pixels in most of Gumin's WFC examples, the set of game map tiles in *Caves of Qud*, the collection of modules of 3D geometry in *Bad North*, the lexicon of words in O'Leary's *Oisín*, and so forth. Generalized, tiles are discrete sets of items that can be uniquely recognized at each location in the input and for which new outputs can be assembled from a grid (or graph) of tile identifiers.

Tiles in the solver don't have to directly correspond to the tiles as displayed in the final result. Some WFC generators, such as the one used in *Bad North*, support tiles that span multiple cells. This behavior can also be implemented by having invisible tiles with adjacency constraints that require them to be the only things in the multi-tile

[6]"Content-agnostic algorithm for placing tiles. Heavily based on the work of @ExUtumno. Basically a Sudoku-solver on steroids." `https://twitter.com/OskSta/status/784847588893814785`

[7]`https://github.com/lamelizard/GraphWaveFunctionCollapse`

[8]"last one for tonight, a 3 second loop!" `https://twitter.com/MattRix/status/872674537799913472`

[9]`https://github.com/mewo2/oisin`

region.

9.2.2.1 Learning Constraint Solving rules via Machine Learning

Because the constraint solver is the heart of WaveFunctionCollapse, the adjacency constraints are critical data and the adjacency learning stage is a vital (but distinct) part of the overall generator. In the most extreme simplification these can be the literal input tiles, with hand-written adjacency data. However, rather than operating on literal tiles, WFC typically operates on higher-order data about the tiles plus their local context: Gumin's name for this data is *patterns*. Other strategies for learning adjacency include the approach used in Bad North, which takes the 3D geometry modules (exported from a 3D modeling program) and classifies them by whether the profiles match. Profiles are the (polyline) cross-sections of 3D meshes sliced at grid cell boundaries.

Gumin's OverlappingModel describes patterns as an $N \times N$ region of tiles (where N is typically 2 or 3)[10] that act as the context for a given tile. Therefore, a single tile can be in multiple patterns. Valid pattern-region adjacencies are when the intersection of two pattern-regions are identical when overlapped with a given offset in space (e.g. pattern 8 overlaps with pattern 57 in direction (0,1) in Fig. 9.2). Gumin's implementation represents this as a Boolean matrix with the shape *pattern* $\times$ *pattern* $\times$ *offset direction* which is optimized for quick lookups in the solver. However, when pre-processing the patterns the list of adjacency tuples is often easier to work with.

[10]Large values of N rapidly increase the complexity of the generated patterns.

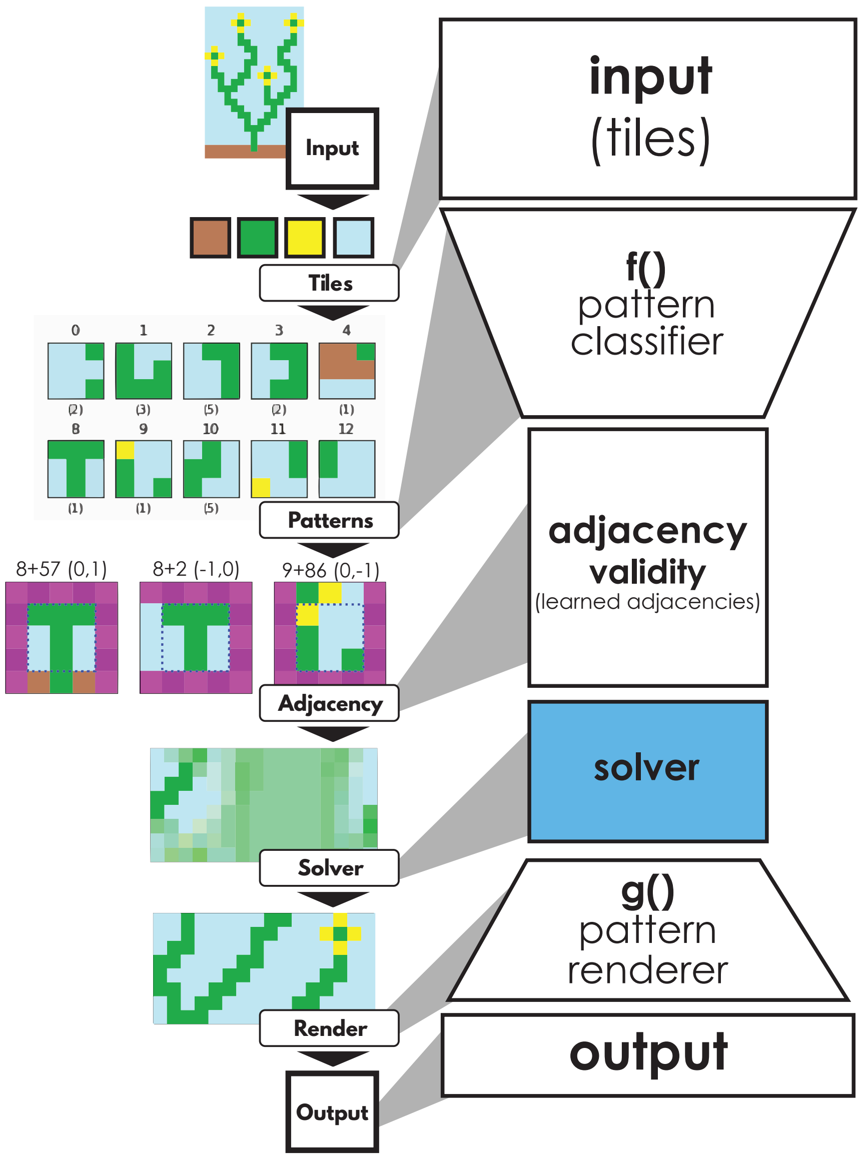


Figure 9.2: Diagram of the WFC generative pipeline, using the overlapping model to automatically determine allowed adjacencies between tiles. Left column shows selections of data at each step in the pipeline, right column shows how the functions in the pipeline are related. Starting with a source image, it is (1) subdivided into tiles; (2) tiles in source image are classified into patterns via a convolution of their local neighborhood; (3) a matrix of pattern adjacency constraints is constructed based on how patterns overlap with neighboring tiles, the image here shows how the two neighboring patterns overlap on a specific dimension; (4) the constraint solver generates a new configuration of patterns; (5) the pattern renderer converts the patterns into tiles (or pixels or whatever other representation is desired); and (6) finally, resulting image is output.

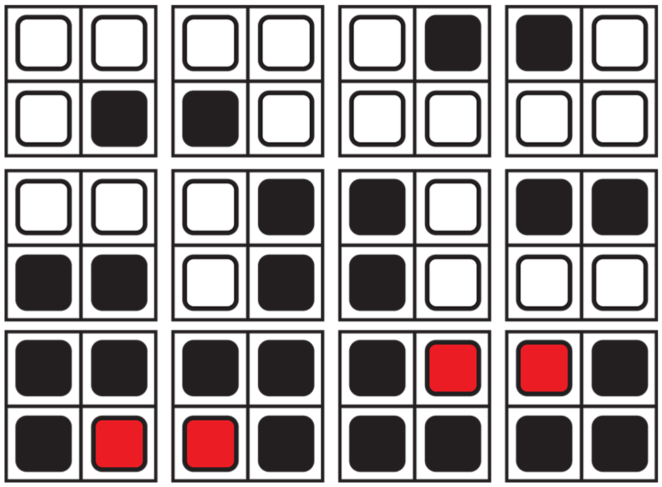


Figure 9.3: The twelve 2×2 patterns extracted from the RedMaze.png image sample.

Working with patterns is technically a generalization of tile relationships (representing each tile as a set of all allowed combinations of tile-plus-neighbors). However, the abstraction gives us a powerful way to compactly translate images into constraint rules.

As can be seen in Fig. 9.3, when $N = 2$, the maze sample contains twelve unique patterns. Four with a single black pixel, four with two black and two white, and four around the red pixel. The red and white pixels are never next to each other in the source image, so there is no pattern with that combination. Note that the sampled image is periodically tiling. This is the default setting of the original WFC implementation; it is an optional detail that allows one image to compactly specify tile relationships without being concerned about edges.

The analysis can be considered a form of machine learning [187]. Whether one pattern can be placed next to another with a given offset can be decided by a classifier that accepts two patterns and outputs a Boolean judgment. This classifier is fit to agree with the adjacencies demonstrated in the source image. One simple representation of this classifier is simply a list of allowed pattern pairings.

9.2.3 Constraint Solving

The core of WFC is the constraint solver. This can be implemented with an existing constraint solver, such as Clingo [186]. However, Gumin's reference implementation is a custom-written probabilistic constraint solver in C#, which is tailored to the requirements of WFC constraint solving. The following description breaks down how each part of the solver in the reference implementation works, as well as some remarks on variations.

9.2.3.1 Initialization with Clear()

The state of the solver is primarily captured in one data structure: the *wave matrix*, representing which patterns are still valid for each cell by a Boolean value. The *adjacency matrix*, representing the patterns that are valid for each edge per pattern, is unchanging in the solver. Optionally, an auxiliary table of counters may avoid having to recalculate the number of remaining patterns and their sampling weights. At the start of solving, every entry in the wave matrix is cleared to a value of `true`.

One metaphor is to consider the wave matrix as a chess board, with puzzle pieces stacked in each space that represent the patterns. At the start, each space on the board has a stack of all the patterns. *Propagation* is the process of removing pieces that no longer fit with any of their potential neighbors. *Observation* is the process of removing all but one of the puzzle pieces. If there is an empty square, a contradiction has occurred and the current state is invalid. The goal is to get the chessboard into a state where every square has exactly one puzzle piece.

9.2.3.2 Removal of alternatives with Ban()

An operation that is shared by both observation and propagation is to remove patterns from the domains of the cells. The reference implementation has a separate function for this. It takes a grid location and a pattern and does the bookkeeping to remove the pattern from the constraint solving domains. The wave matrix entry for the location and pattern are set to false, the pattern is removed from the edge compatibility, and the entropy values are decremented. The neighboring nodes are added to the stack for updating during the propagation phase.

9.2.3.3 Choice of Design with Observe()

The purpose of Observe() is to select a decision variable and decide its value. The reference implementation uses a heuristic that chooses the most constrained cell. The cell with the tightest or smallest domain after propagation has the lowest entropy, with ties broken at random. There are two termination cases: first, if any of the cells have zero remaining patterns, the search has dead-ended with a contradiction. Second, if all of the cells have an entropy of one (i.e., there is exactly one pattern for each node, which is the minimum non-contradictory state) we have found our solution.

The heuristic of selecting the most constrained variable, or equivalently the variable with minimum remaining values (MRV), is well-known in the constraint solving literature [299, Chap. 6].

The strategy of selecting the location with the lowest non-zero entropy (or minimum remaining values) might seem arbitrary at first. If we want to optimize our chances of uncovering a total assignment without restarting or backtracking, we should make choices that maximize the number of total assignments consistent with our choices so far. This avoids ruling out legal (though potentially extremely rare) total assignments under the assumption that they are distributed amongst other total assignments. If the number of remaining total assignments is approximated as the product of the size of the domains of the unassigned variables (in other words, assuming the remaining choices are independent), then assigning the location with the smallest domain (lowest entropy / minimum remaining values) maximizes the value of the product after the assignment. To make another loose physics connection (this time to statistical mechanics), Gumin's (least) entropy heuristic could be interpreted as an intent to maximize the entropy of a uniform distribution over possible completed designs.

Once a cell is chosen, one pattern needs to be selected from the domain. This is done through weighted random sampling, with the weight derived from the frequency that the pattern appears in the input training data image. This implements Gumin's secondary goal for local similarity: that patterns appear with a similar distribution in the output as are found in the input [145]. All non-selected patterns are then removed from the cell's domain. The reference implementation does this with the Ban() function which also adds the neighbors onto the propagation stack and takes care of the entropy bookkeeping.

The first observation selects between all of the possible patterns, but the subsequent observations tend to look at vastly fewer options. The exact number is heavily dependent on the input image, but it is typical that the algorithm only chooses between two or three remaining valid patterns in a domain.

9.2.3.4 Resolving implications with Propagate()

Once an observation has been performed, the neighboring cells need to be updated. Updating the domain of one cell implies the need to update the adjacent cells, propagating the implications of the previous observations via the Ban() function.

Even within Gumin's reference implementation of WFC, the propagation approach has changed over time. Early pre-release implementations of the propagation function used belief propagation, but was later changed to "constraint propagation with a saved stationary distribution" [145] for performance reasons. A further post-release optimization changed the reference implementation of WFC to execute propagation with a stack-based flood fill, incorporating performance improvements introduced by Fehr and Courant [106, 144].

Propagation can result in the domain of a cell being reduced to one pattern, completely resolving that cell. The distribution of which cells are resolved by propagation and which through observation varies based on the input image, and visualizing this can be useful for debugging.

The observation-and-propagation loop demonstrates that WFC is a family of algorithms that use constraint solving as the core generation algorithm. Indeed, Gumin occasionally describes the algorithm as a constraint solver.[11] It uses the minimum remaining values (MRV) heuristic to select the variable to decide next. For decisions, it uses the heuristic of choosing patterns according to their distribution in the original image. An alternative to this heuristic would be to use the well known least constraining value (LCV) selection heuristic [299]. (LCV can also be motivated by the maximum entropy principle.) However, it is difficult to predict the implications of this heuristic choice for the purposes of content generation. The topic of sampling from combinatorial spaces with statistical uniformity guarantees is surprisingly subtle [131].

9.2.3.5 Contradictions

Unlike many constraint solvers, the reference implementation of WFC does not employ backtracking. When it encounters a contradiction, it restarts from the beginning, giving up if it fails too many times (by default, the limit is ten attempts). The nature of the generation task makes this less of an issue than it would be for a general constraint solving problem: the patterns are likely to fit together (at least in the same way they did in the input image), ideally in many different configurations. So for most input images in the sample set at the sizes the solver is trying to generate contradictions are very rare.

It is possible to construct an input image that forces contradictions: the sample

[11]"I can help with the grasp part. WFC is basically a constraint solver. You start with everything unknown and when possible..." `https://twitter.com/ExUtumno/status/793601984800624640`

set has the example of a wrapping chessboard with odd length and width, which is, of course, impossible to solve. It is also possible to construct inputs that are trivially perfect.[12]

WFC encounters contradictions more frequently with more restricted tilesets and larger output spaces. Contradictions also, as discussed in our experiments below, can become more frequent when additional global constraints are introduced. At this point, the addition of backtracking becomes more important, as does the ability to distinguish between an impossible problem and a merely complicated one.

9.2.4 Rendering

Once the constraint solver has discovered a solution, it is represented as a grid of pattern identifiers. Therefore, one final step needs to be performed to translate the patterns back into tiles. In the reference implementation this is straightforward: because the patterns overlap, each cell resolves to exactly one tile. In most WFC variations, tiles are simply reproduced in the form they appear in the source image.

However, variations are possible. One variation in the reference implementation is used to render partial solutions: because a partially-solved constraint problem has more than one pattern per cell, the visualization for the algorithm in progress averages the colors of the tiles derived from the patterns in the domain for each cell.

An implementation that uses a more interesting representation for the tiles

[12]Gumin gives an example of an "easy" tileset: "the unrestrained knot tileset (with all 5 tiles being allowed) is not interesting for WFC, because you can't run into a situation where you can't place a tile" and without "special heuristics" the length of correlations in the generated image quickly fall [145].

might include additional processing in the rendering step. For example, if the tiles are part of a 3D landscape generator, the number of trees added on this particular tile can be additionally informed by how many other forest tiles are nearby, quite apart from the encapsulated WFC step in the generation pipeline. A generator that has a coastline might use only water and land tiles in the WFC patterns, and only add the coastal transition sub-tiles as part of a post-processing step. This possibility is examined more deeply in one of the experiments in the next section.

9.2.5 Visualizing WFC

One of the factors that led to WFC going viral[13] are the visualizations of the solving process. The way these animations visualize modifications to partially-completed design immediately sets this generative method apart from ones based on a generate-and-test paradigm, where only complete designs are considered during execution.

In addition to the visualizations of the intermediate states of the solving (showing the averaged possibility space of each cell) other useful visualizations can help us understand what is happening or has happened during the generation process. I present here a visualization that I call a *crystal growth diagram* (see on the bottom row of Fig. 9.5): a map showing the order in which the solver resolved the final contents of each cell in a single image rather than an animation. Such diagrams record the incremental growth of a design without the use of animation in the same way as the patterns of colors apparent in the cross-section of a geode.

[13]Starting with the tweet "Procedural generation from a single example by wave function collapse" `https://twitter.com/ExUtumno/status/781833475884277760`

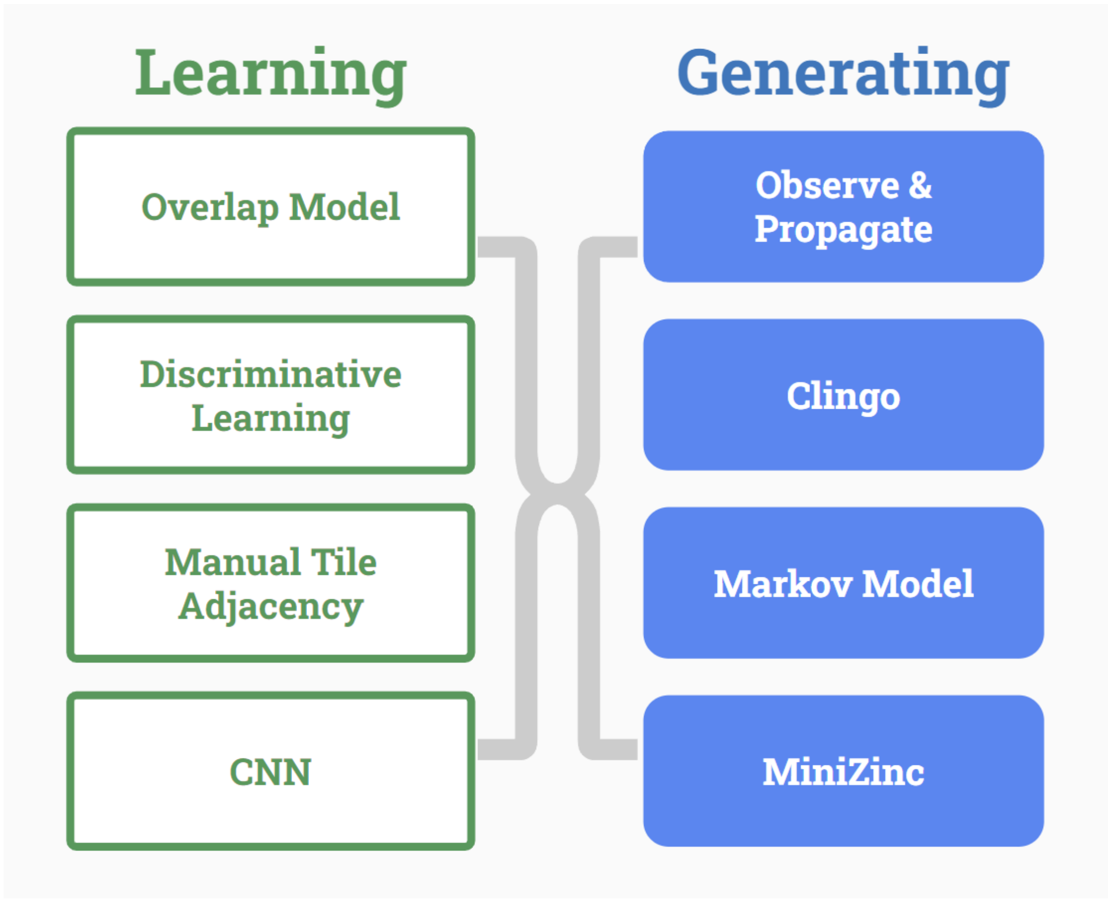


Figure 9.4: Any of the different implementations of the learning component can be mixed-and-matched with any of the different implementations of the solver and it is still WaveFunctionCollapse.

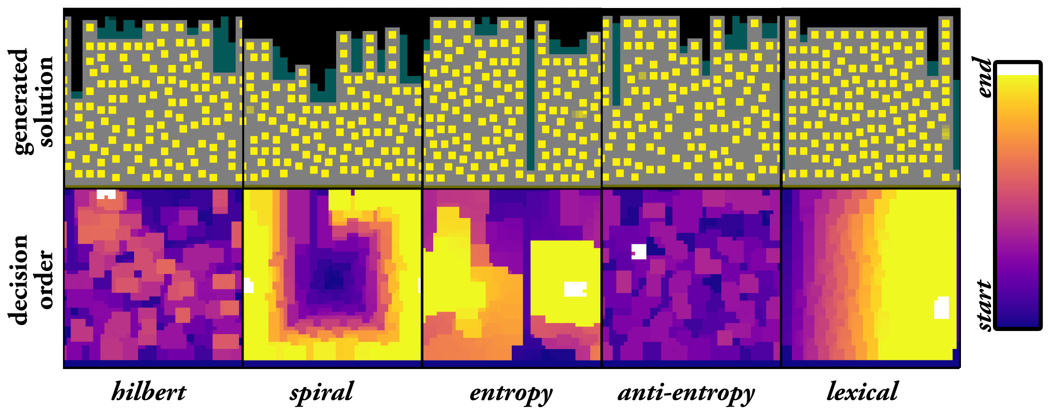


Figure 9.5: Visualizing the order in which design decisions are made when using different heuristics. Final results are shown on the top row while crystal growth diagrams on the bottom show the order in which the result was assembled. Note that with strongly anisotropic pattern adjacencies (e.g. a side-view of a skyline) some location heuristics (such as *lexical*) tend to produce more clustered results (in this case, placing nearly all of the roofs of the buildings near the top of the generated image) which may or may not be aesthetically desirable.

Other visualizations I have found useful for debugging are tracking whether a cell was resolved with propagation or observation; the number of patterns an observation selects from; and the number of remaining possible patterns for a cell. These visualizations have revealed that most cells are resolved via propagation, and further that the cells with the least remaining patterns (those likely to be selected for observation next) are usually adjacent to cells that have been recently resolved.

9.2.6 Implementations in General-Purpose Constraint Solvers

In our initial investigations of WFC in 2017, we re-implemented the core of WFC in the paradigm of answer-set programming. Per the design space modeling approach [321], this answer set program functions as a compact definition of valid designs without referring to a specific algorithm for sampling those designs. MiniZinc is another constraint modeling language. Unlike typical ASP, it is common for MiniZinc models to include additional clues to the underlying constraint solvers about how the search should proceed, and our MiniZinc implementation takes advantage of this. In both cases, WFC can be expressed as rules for the constraint solver. One drawback of this is that using WFC for procedural generation is searching the possibility space for any single solution, while general-purpose constraint solvers are generally designed to find an optimal solution. Additionally, WFC scenarios typically have many easily found solutions. General-purposes solvers become more viable as the complexity of the constraint-solving increases (for example, by adding more global constraints).

9.2.6.1 MiniZinc

MiniZinc is another constraint modeling language. In contrast to typical uses of ASP, it is common for MiniZinc models to include additional clues to the underlying constraint solvers about how the search for solutions should proceed. Our MiniZinc implementation models the generation task while also declaring the intent to use a minimum-remaining-values heuristic to select the next variable and a random heuristic to decide which value it should take (approximately expressing Gumin's entropy heuristic and weighted random sampling strategy).

9.3 Experiments

There are many implementations of WFC (the reference repository links to over 20 publicly available implementations in more than a dozen programming languages). While most function similarly, there are many variations already extant: adding a time dimension;[14] edges for rhymes and metre [262][15]; using backtracking; or adding global constraints (e.g. testing if a level is traversable from entrance to exit). This section experimentally probes the importance of design decisions in Gumin's original WFC, aiming to separate the essence of the idea from its first implementation. In particular:

- Is the *entropy* location heuristic necessary?
- Is the *weighted* pattern heuristic necessary?
- Is it safe to run WFC *without backtracking* when plausible global constraints are added?
- Does WFC need to operate on *human-curated* tiles?

9.3.1 Location (Selection) Heuristics Experiment

The heuristic Gumin used for selecting the location of observations in WFC is partially inspired by observing humans drawing: "Why the minimal entropy heuristic? I noticed that when humans draw something they often follow the minimal entropy

[14]@MattRix, Jun 8, 2017: "yep exactly! :) it has 3 dimensions, but the trick is that it has 14 edges so it can check forward & backward "diagonally" in time" `https://twitter.com/MattRix/status/872785320445706241`

[15]`https://github.com/mewo2/oisin`

heuristic themselves. That's why the algorithm is so enjoyable to watch."[16] However, we know that other heuristics are possible: several of the other implementations of WFC use different ones. Therefore, in my first experiment here, I aim to understand the importance of Gumin's entropy heuristic. I do this by comparing multiple heuristics in our reconstructed Python implementation[17] which is equivalent to Gumin's in terms of the state of the wave matrix after each observe–propagate cycle.

For all of the experiments, I used the reference implementation's selection of example source images. These define 53 different settings for testing the overlap WFC model. The heuristics tested were as follows:

entropy
Original entropy calculation: select the node with the most constrained domain.

anti Opposite of entropy, select the node with the least constrained domain, breaking ties randomly (Included for a comparison as the most degenerate strategy).

lexical Select the first unsolved ($domain > 1$) node in top-to-bottom, left-to-right lexical order.

random
Select a random unsolved node.

spiral Select the first unsolved node found by traversing a spiral path outward from the center.

Hilbert
Select the first unsolved node along a space-filling curve.

Rather than measuring wall-clock time (which is likely to vary between implementations), we measure the number of choices (observations) that the solver makes

[16]`https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md`
[17]`https://github.com/ikarth/wfc_2019f`

against the number of test scenarios that each heuristic successfully completes. A good, fast heuristic is one that makes fewer choices (resolving faster) while minimizing restarts.

My hypothesis was that the main factor in the entropy heuristic's success was that it kept the search space as the frontier of a growing solved continent, rather than an archipelago of islands that might contradict when they grew enough to encounter each other. I expected the heuristics to fall into two broad groups: entropy, lexical, spiral, and Hilbert tend to grow contiguous regions; while random and anti-entropy created discontiguous chunks.

As expected, the degenerate case of anti-entropy found the fewest solutions (Fig. 9.6). While the heuristics that sampled the entire unsolved space were often faster when they reached a solution, there were significantly fewer solutions found. In contrast, the heuristics that focused the search on a growing solved region performed similarly to each other. Surprisingly, Hilbert performed worse than expected, most likely due to the space-filling curve not interacting well with the dimensions of the output. These findings about which order is best to assign tile codes in a grid roughly confirm the conclusions from a recent paper on transformer-based image generation [104], namely that new tiles should be chosen very close to previously chosen tiles and that a simple lexical ordering (e.g. "row-major" in their terms) was sufficient.

However, when choosing an heuristic, the aesthetic dimension should also be considered. For example, when using a training image with strongly anisotropic details, such as a side view of a city skyline, the solutions found tend to be very dependent on the location heuristic used (Fig. 9.5). Because the building roofs can only have sky above

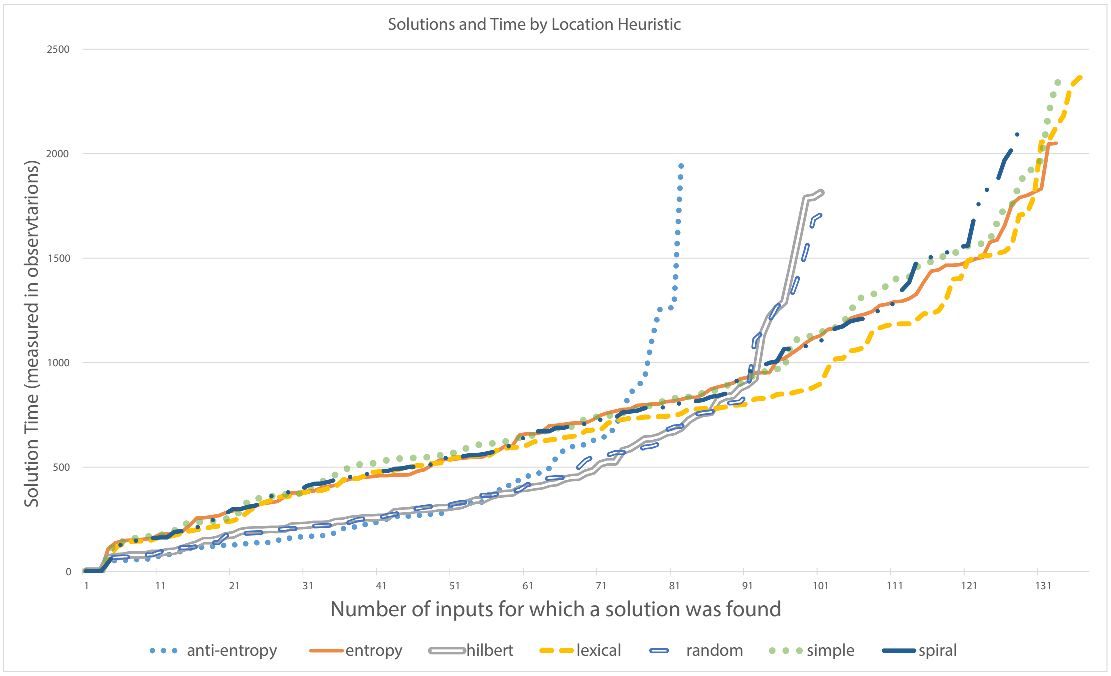


Figure 9.6: Cactus plot of location (selection) heuristic experiments. Lines reaching further to the right indicate more test scenarios solved, lower lines indicate that fewer choices (observations) were required. The choice count for a solution include any restarts required because of contradictions. The default limit of 10 restarts was used. There are three clusters of performance: anti-entropy had the fewest solutions, Hilbert and random had a few more, and the rest performed comparably. In general, discontiguous location heuristics tend to find solutions slightly faster but are more likely to encounter contradictions.

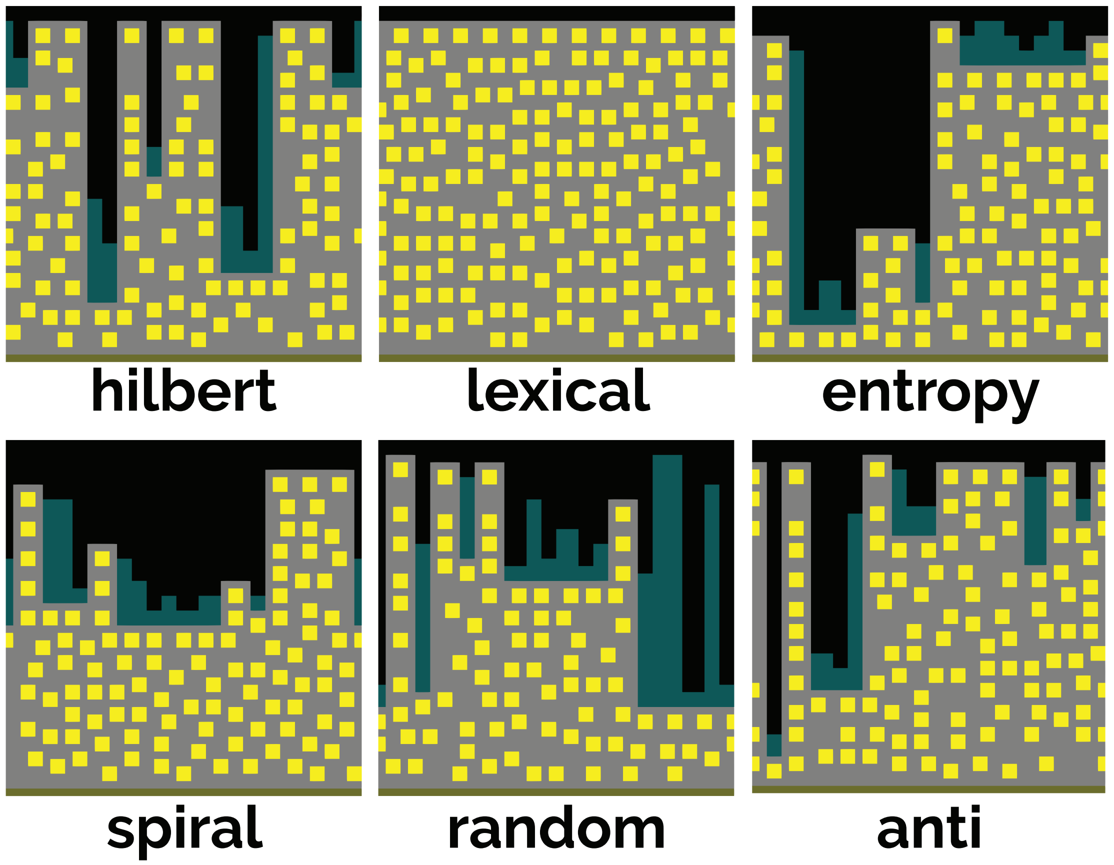


Figure 9.7: Comparison of typical results of different location (selection) heuristics. Most notably, with strongly anisotropic pattern adjacencies—such as used in a side-view of a skyline—lexical location heuristics with a fixed starting location tended to exhibit bias due to early pattern choices dominating the expressive range, which is often not aesthetically desirable.

them, the lexical ordering leads to the roofs being placed earlier than would occur with a random starting location. In this case, lexical ordering strongly biases the generation, reducing the expressive range. While this lexical anisotropy may be a valid aesthetic choice for some purposes, the entropy heuristic performs well on both the performance and aesthetic dimensions, and when in doubt should be the default.

9.3.2 Pattern (Decision) Heuristics Experiment

The second major heuristic is used to decide a value (pattern) for each variable (grid location).

weighted

The reference implementation uses a random sample weighted by the frequency that the pattern appears in the source image.

random Uniform random sampling from the domain

rarest Choose the least-used pattern; defined as the pattern that appears with the lowest frequency in all the remaining domains

common

Choose the most-used pattern; defined as the pattern that appears with the highest frequency in all the remaining domains

lexical Choose the pattern with the lowest array index

Rarest and *common* frequently run into contradictions. *Lexical* performs surprisingly well, depending on the characteristics of the input. But only *weighted* and

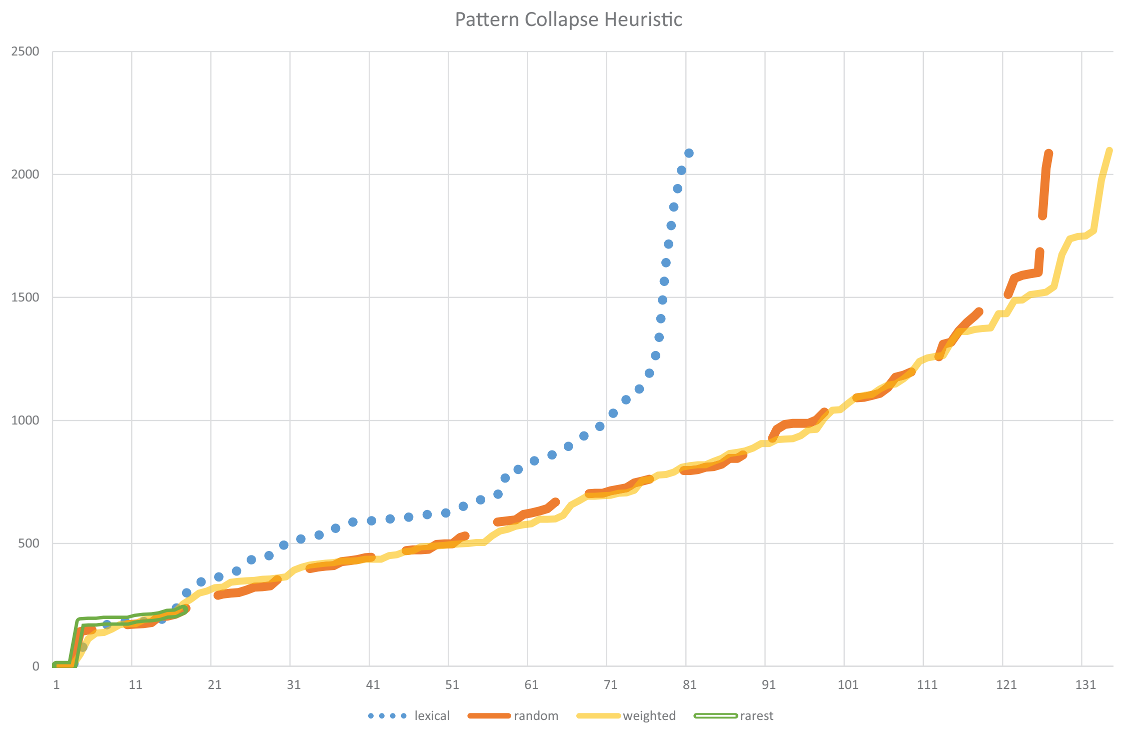


Figure 9.8: Cactus plot of pattern selection heuristic. Weighted performs slightly better than uniform random, while lexical selection performs notably worse, as expected.

random find enough solutions to be generally viable (Fig. 9.8). The "local similarity" in Gumin's "locally similar to the input bitmap" includes the "Weak C2" characteristic: [145]

> Distribution of NxN patterns in the input should be similar to the distribution of NxN patterns over a sufficiently large number of outputs. In other words, probability to meet a particular pattern in the output should be close to the density of such patterns in the input.

This is satisfied stochastically by the weighted pattern heuristic: it is likely to sample patterns with the desired characteristic frequency. In practice, many input images are relatively homogeneous and this matters more for scenarios with rare but noticeable patterns.

9.3.3 Backtracking Experiment

A surprising feature that the original WFC algorithm lacked was backtracking. Backtracking is a common feature in constraint solvers [299, Chap. 6], but WFC performed quite well without it, instead opting for a brute-force restarting approach. Some implementations of WFC added backtracking.[18] The question remained: how effective is backtracking for typical WFC problems? There was expressed interest in the wild in adding global constraints.[19] My hypothesis was that generation with global constraints would be more likely to benefit from the ability to backtrack. To test this, I implemented an "allpatterns" global constraint, which required that every pattern found in the input should be used at least once in the output (Fig. 9.9).

For the test scenario set, backtracking had a minor benefit when only local (adjacency) constraints were used. This is consistent with the general performance of WFC: using constraint solving for procedural generation allows for much more leeway in solution-finding compared to general-purpose constraint solving. With WFC, contradictions are often clashes between deeply incompatible regions in the solution, so shallow backtracking does not have a dramatic effect on the result. In contrast, adding a global constraint has a noticeable impact on performance. Solutions consistently take longer, with significantly more failures. In this case, backtracking makes the global constraint performance comparable to the local constraint performance.[20] This confirms our ear-

[18]Including Oskar Stålberg's implementation and the PICO-8 implementation by Rémy Devaux: `https://trasevol.dog/2017/09/01/di19/`

[19]"No, it's a very good question, I think about global constraints like connectivity a lot. Thanks!" `https://twitter.com/ExUtumno/status/760235500858900480`

[20]This is partially because of the rapid detection of when the global constraint has been violated. Global constraints that require more computation to verify will be correspondingly slower, and global

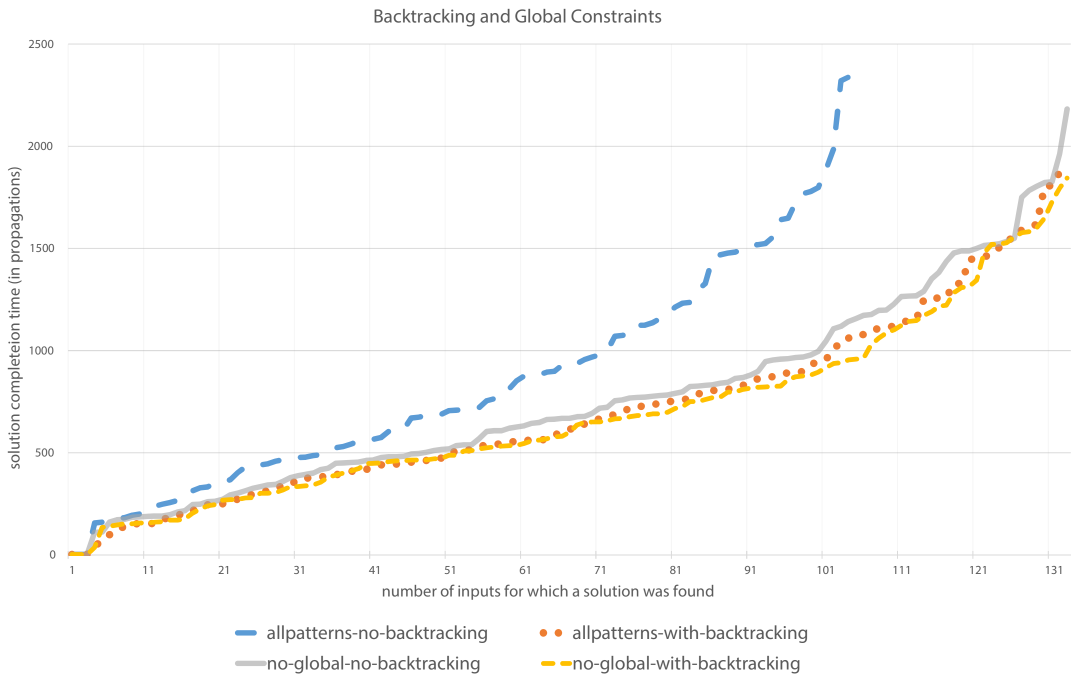


Figure 9.9: Cactus plot for backtracking experiments. With only local (adjacency) constraints, backtracking improves performance slightly. Introducing a global constraint without backtracking significantly increases the difficulty (solutions take longer to find and/or fail more often). However, a global constraint with backtracking has comparable performance to non-global constraint scenarios.

lier experimental results with global constraints and backtracking using an ASP-based rational reconstruction of WFC [186].

Importantly, despite the simplicity of adding backtracking to an existing WFC implementation, backtracking makes the search algorithm *complete*: it will always find a solution if one exists, given enough time. Whether a method *may fail* has been a notable feature examined in other example-driven generative methods research [217].

9.3.4 Learned Classification and Rendering Experiment

In Chapter 11, I discuss an experiment in combining WFC with deep learning to generate novel tilesets. WFC isn't constrained to

These results demonstrate that WFC does not inherently require a carefully curated tileset. The strategy of using a learned latent tile vocabulary for high-resolution image synthesis here parallels recent work in computer vision (where a transformer-based model replaces WFC for the purposes of generating fresh tile grids with spatial coherence) [104]. Computer vision techniques have previously been used in other work on map generation from images and videos (e.g. in the literature of PCGML [344]), but the focus has been on maps composed of a small vocabulary of identically rendered tiles. In the demonstration in Chapter 11, the value of context-dependent rendering is shown in how local combinations of just 16 latent tile codes can be used to express many more visually-distinct image patches.

constraints that have more long-range implications will require more backtracking to be successful.

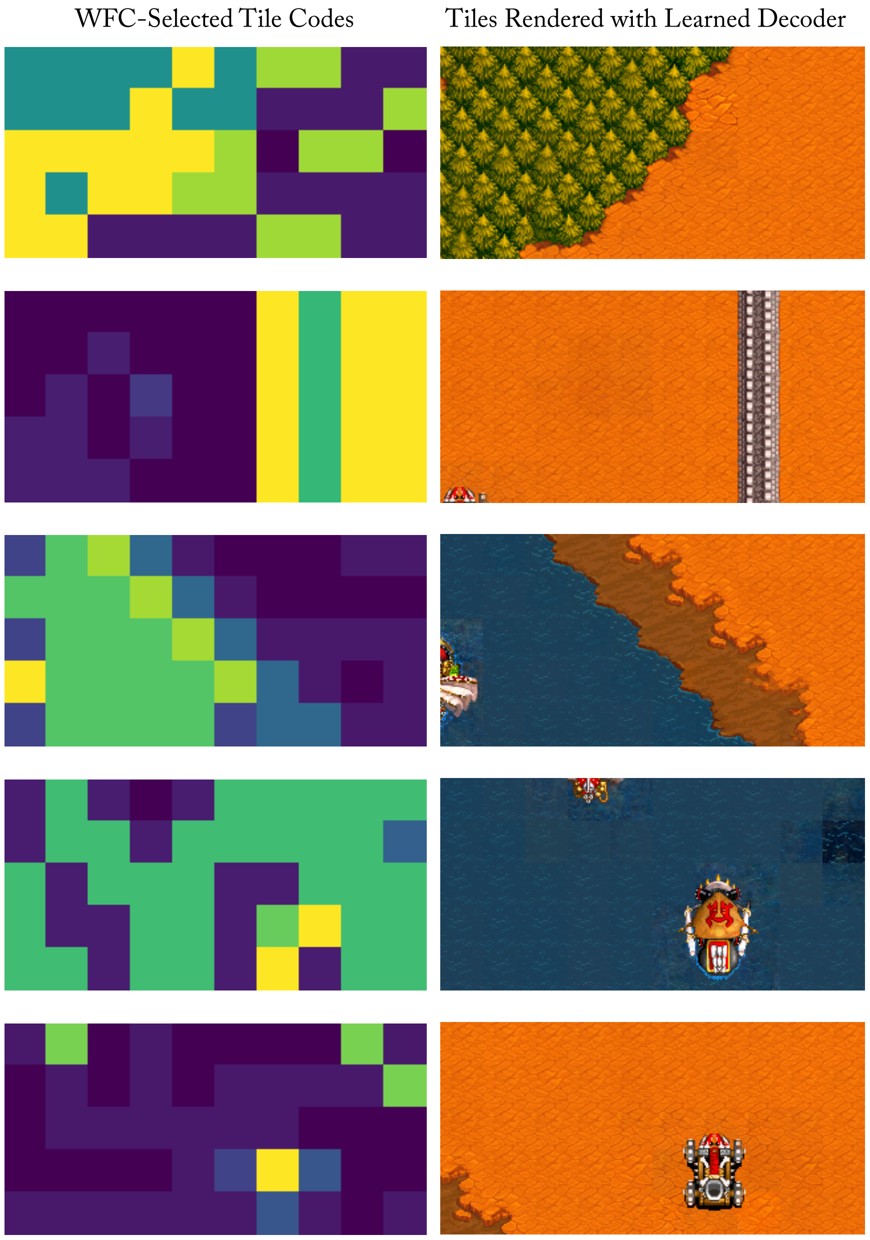


Figure 9.10: Novel WFC-generated WarCraft map segments. Images on the left visualize the grid of tile codes (referencing a 512-dimensional vector codebook with 16 entries) output by the solver core of WFC, and those at the right show the result of passing those grids through the learned neural decoder. These five samples were chosen from a larger batch of 20 unconstrained outputs in order to highlight the variety of terrain transitions and building or unit types.

9.4 Tracing WFC in the Wild

9.4.1 Game Development

Within a day of the September 30th, 2016 release of WaveFunctionCollapse to the public [140], other developers were actively experimenting with it in the wild. Spreading through social media, particularly via images and animations on Twitter, WFC soon had re-implementations in many environments, including multiple ones for the PICO-8 fantasy console.[21]

One game developer who has contributed to the popularization of WFC is Oskar Stålberg. A technical artist who previously worked on *Tom Clancy's The Division* [159], Stålberg was among the first to start generalizing WFC, extending it with other tile shapes,[22] 3D meshes,[23] performance optimizations,[24] and adding backtracking.[25] In May 2017, as part of a talk about his approach to procedural generation, Stålberg released a "small browser demo"[26] to illustrate how his version of the algorithm works under the hood [339]. Notably, Stålberg's WFC started as a clean-room implementation based on the animations rather than on the original code.[27] Stålberg

[21]By Joseph Parker `https://twitter.com/jplur_/status/873551783347589120` and TRASEVOL_DOG / Rémy Devaux `https://trasevol.dog/2017/06/19/week60`

[22]"Content-agnostic algorithm for placing tiles. Heavily based on the work of @ExUtumno. Basically a Sudoku-solver on steroids." `https://twitter.com/OskSta/status/784847588893814785`

[23]"More procedural tile placement. Now in 3D. Algorithm inspired by the work of @ExUtumno" `https://twitter.com/OskSta/status/787319655648100352`

[24]"It's getting faster (mostly due to bitwise operations). Actual speed depicted below" `https://twitter.com/OskSta/status/794993371261665280`

[25]"I gave it an extra difficult tileset to work with to make sure it can repair itself when it has screwed up" `https://twitter.com/OskSta/status/793806535898136576`

[26]"I built a small browser demo to help explain how the WFC algorithm works. Give it a go: http://oskarstalberg.com/game/wave/wave.html …" `https://twitter.com/OskSta/status/865200072685912064`

[27]"It was hard to crack your computer science lingo about collapsing wave functions, so I basically

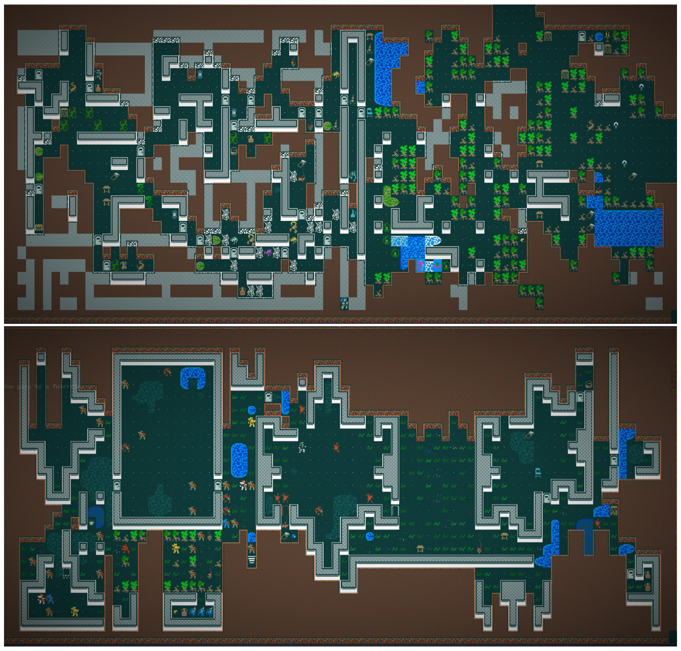


Figure 9.11: Two different historical site levels in *Caves of Qud* generated via WaveFunctionCollapse. Copyright 2016 Freehold Games. Used with permission.

would go on to use his implementation of WFC in the viking invasion game *Bad North*, released in 2018.

One commercially-released indie game that uses WFC is *Caves of Qud* [139], which added partially-WFC-generated villages in July 2018. *Caves of Qud* is a rogue-like developed by Freehold Games that is currently in early-access release.[28] One of the developers, Brian Bucklew, started experimenting with using WFC for level generation.[29] *Caves of Qud* uses WFC as part of a pipeline, adding additional pre-generation constraints, using WFC multiple times with different input images, and making further

just looked at your gifs" `https://twitter.com/OskSta/status/784850280093478912`

[28]though the 1.0 release is currently in beta testing, as of November 2022.

[29]"The peaceful gardens of Inner Aarranip. A Caves of Qud dungeon generated via https://github.com/mxgmn/WaveFunctionCollapse ...based synthesis." `https://twitter.com/unormal/status/805987523596091392`

changes to the map after the generation is run.[30] One of the benefits of WFC that *Caves of Qud* has demonstrated is that the simple inputs mean that it is much easier for the entire team to experiment with the generator.[31] The *Caves of Qud* developers noticed two drawbacks of WFC: isotropy and overfitting [45]. While WFC can capture long range correlations, there are limits and large images tend toward isotropy and homogeneity at lower frequencies. Their solution to this was to partition the space with other algorithms and only run WFC on a subset of the level. Likewise, adding more detail to the small training image risked over-constraining the results, as the training overfitted to the image. Their solution was to avoid putting too much detail in the training image. Instead they add detailing (such as extra doors) using other approaches later in the pipeline.[32]

WaveFunctionCollapse in *Caves of Qud* is an addition to an existing level generator. While *Bad North* also uses WFC as part of a pipeline, the *Bad North* island generator would not exist without WFC. *Bad North* islands are constructed out of 3D modules, designed in Maya and exported into Unity with a process that calculates which modules have matching geometry along their edges. These “profiles” are used to define the adjacencies: modules with matching profiles are allowed to be adjacent. Some larger modules take up multiple spaces. Variation across islands is achieved by changing the set of modules allowed for an island, thereby creating regions of similar islands by having

[30]`https://forums.somethingawful.com/showthread.php?threadid=3563643&userid=68893&perpage=40&pagenumber=23#post467126402`

[31]“Ha! I'm experimenting with the WFC map generator @unormal just added.” `https://twitter.com/ptychomancer/status/805964921443782656`

[32]WFC can also be run twice,

them use similar sets of tiles and allowing a progression to be created from the input to WFC. The islands are constructed on three-dimensional 11×11 grids (with some degree of vertical space) and some additional constraints, the most important of which is a connectivity local constraint: the island needs guaranteed paths from the Vikings on the beach to the houses the player is defending.

9.4.2 Artistic

In addition to level design, WFC has been applied to other kinds of content. One of the most unexpected was developed by Martin O'Leary, a glaciologist who also makes "weird internet stuff" [263] including twitter bots and generated travel guides. Inspired by WFC, O'Leary created a poetry generator called Oisín which enforces rhyme/meter constraints to construct sonnets,[33] limericks,[34] and ballads.[35] In personal correspondence with us, O'Leary explained that, "I treat syllables as the basic unit, so each 'tile' is a sequence of syllables (tagged with the word/position it comes from)." This is put into a 1-dimensional WFC sequence, together with "some extra long-distance constraints induced by rhyme, meter, etc." O'Leary has released the source code [262].

Oisín also demonstrates that, while WFC is typically used on a rectangular

[33]"Using @ExUtumno's "wavefunction collapse" algorithm to enforce rhyme/meter constraints in text (Alice in Wonderland as Shakespearean sonnet)" `https://twitter.com/mewo2/status/789167437518217216`

[34]"Also, Pride and Prejudice as a limerick. Turns out the limerick constraints are much harder to satisfy than sonnets, which are easy to write"
`https://twitter.com/mewo2/status/789177702620114945`

[35]"Moby Dick in a conveniently singable ballad form, thanks to WFC"
`https://twitter.com/mewo2/status/789187174683987968`

grid, the algorithm has already been extended to graphs. Indeed, Oskar Stålberg demonstrated WFC on the surface of a sphere tessellated with triangles, Github user "Boris the Brave" made an implementation that supports both hexagonal grids and global constraints,[36] and Florian Drux implemented WFC for arbitrary graphs as GraphWaveFunctionCollapse.[37]

9.4.3 Academic

On the research side, WFC has been incorporated into several avenues of research. In "Procedural Content Generation via Machine Learning" (PCGML), Summerville et al. described WFC in relation to Markov random fields and constraint propagation [344].

In "Addressing the Fundamental Tension of PCGML with Discriminative Learning", I use WFC to discuss the tension between having to author enough training data and saving effort. I also discussed how the overlap model's pattern classifier/renderer uses machine learning (similar to autoencoders) and how it can use multiple input sources as part of a mixed-initiative design approach [187].

WFC has also been applied to other research problems and generators. Khokhlov, Koh, and Huang developed a voxel synthesis approach based on WFC and Merrell's texture synthesis work, taking advantage of WFC's ability to learn from single-input models [194]. Oliveria et al. used WFC to generate a city as part of an experiment into mixed

[36]`https://boristhebrave.github.io/DeBroglie/`

[37]`https://github.com/lamelizard/GraphWaveFunctionCollapse` and `https://github.com/lamelizard/GraphWaveFunctionCollapse/blob/master/thesis.pdf`

reality cycling environments [264]. Scurti and Verbrugge used WFC as part of a pipeline to generate example-based paths based on non-programmer input [305]. Finally, Snodgrass includes WFC in a theoretical framework of Markov Models in PCG [330].

9.4.4 What led WFC's rapid adoption?

WaveFunctionCollapse spread quickly, being adopted by other developers from almost the moment of its public release. As many researchers face the problem of disseminating their work beyond their immediate academic audience, I believe that it would be valuable to know what led to this rapid adoption. I have identified four possible reasons for this rapid adoption: there were animations of the algorithm in action; the code for the project was available and easy to adopt; artists could use it without understanding the code; and it was at the right time and place to take advantage of network effects and go viral.

Just the animation of the solver working was so compelling on its own that some developers implemented what they thought it was doing without looking at the C# code.[38] In addition, WFC can generate recognizable objects, not just patterns, so the animations were that much more compelling.

The availability of the C# code meant that users who were working in that language could use it right away (e.g. users of the Unity game engine). Further, it does not depend on any particular libraries or frameworks, making it easy to drop into a different project. In presenting it in a way that avoided exclusive reliance on mathematical

[38]"It was hard to crack your computer science lingo about collapsing wave functions, so I basically just looked at your gifs" `https://twitter.com/OskSta/status/784850280093478912`

notation and presenting a computational algorithm computationally, Gumin opened it up to a wider audience. This included artists who didn't want to touch the code at all, instead using interfaces implemented by others to great effect.

Finally, we should not discount that Gumin already had many followers on social media. As many of them were aware of Gumin's earlier texture synthesis projects, they were self-selected to be interested in this new way of generation.

9.5 Conclusion

WaveFunctionCollapse is a family of algorithms that have found widespread use across many spheres. The many implementations of WFC vary in details of the pipeline, but they all use constraint solving for the propagation of relationships and many use some form of machine learning to learn the constraints from tiny amounts of non-programmer input. In contrast to many parameter-controlled generative methods, WFC starts from a high-level example of the desired output and synthesizes the result. The focus on extremely small amounts of example data has made WFC approachable to many outside of academia in a way that the kinds of data-intensive methods examined in the PCGML literature have not [187].

Our experiments demonstrate the relative importance and resource use of various heuristics and the impact of backtracking in the presence of global design constraints. By examining each facet of the pipeline both individually and in context, I demonstrate paths for future research exploration. In particular, through the connection to vector-quantizing models, I have shown a pathway for integrating recent deep learning techniques while still learning from just one (large) example. Future work might examine the possibility of intentionally learning tile classifiers that lead to compactly expressible (e.g. sparse or low-rank) tile-adjacency matrices; or learning rendering functions that take the form of recombinable micro-tiles or other simple representations—so that, after training, no neural networks are involved in rendering (allowing the result to be as widely deployed as current WFC variants).

My explanation of the algorithm is intended to introduce the complex code of the reference implementation in an approachable way. Separating out the different modules in my rational reconstruction enabled me to experiment with heuristics and other aspects of the algorithm's pipeline. Surveying the many implementations and uses for WFC, I have traced how its adoption spread in hobby, academic, and industrial contexts. Following the path of WFC's early adopters, future work should continue to identify and overcome limitations in past variants, demonstrating new ways of using constraint solving and machine learning.

WaveFunctionCollapse presages opportunities for new directions in PCG research. Its influence, already spreading rapidly, ushers us into a new era of PCG that successfully combines techniques that were previously outside the commonly-known PCG sphere.

Acknowledgment

This chapter is based on work that was originally published as “WaveFunction-Collapse is Constraint Solving in the Wild” [186] and “WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning” [188]. Both were co-written with my advisor, Adam M. Smith. The workshop paper was published right before I started the PhD and the journal article finally got officially published this year, so they make a nice pair of bookends for my graduate career.

I would like to thank Oskar Stålberg, Brian Bucklew, Jason Grinblat, Joseph Parker, and Martin O’Leary for their correspondence; and, most importantly, Maxim Gumin for developing WaveFunctionCollapse in the first place.

Chapter 10

Conversational Interfaces with WFC and Discriminative Learning

How does the compositional view help with new applications? (RQ3) Does knowing the structure gain us anything more than the monolithic view could offer us? Given our understanding of how WFC is used in the wild (Chap. 9), how does knowing how the parts of the generative system help us build new generative systems with new interactive affordances?

In this chapter, I explore WFC further, discussing how its modular nature suggested avenues for intervention that lead to proposing new improvements.

10.1 The Machine Learning Content Dilemma

Procedural Content Generation via Machine Learning (PCGML) is the recent term for the strategy of controlling content generators using examples [344] (cf. 3.4.2).

Existing PCGML approaches usually train their statistical models based on pre-existing artist-provided samples of the desired content. However, there is a fundamental tension here: machine learning often works better with more training data, but the effort to produce enough high-quality training data is frequently costly enough that the artists might be better off just making the final content themselves.

Rather than attempting to train a generative statistical model (capturing the distribution of desired content), I focus on applying *discriminative learning*. In discriminative learning, the model learns to judge whether a candidate content artifact would be valid or desirable, independent of the process used to generate it. Pairing a discriminative model with a pre-existing content generator that can accept a validity constraint, we realize example-driven generation that can be influenced by both positive and negative example design fragments. I examine this idea inside of an already-commercially-adopted example-based generation system, WaveFunctionCollapse (WFC) [186].[1] This approach begins to address the fundamental tension in PCGML while also opening connections to mixed-initiative design tools, the source of the design fragments.

Mixed-initiative content generation tools [206] are usually designed around the idea of the artist having a conversation with the tool about one specific design across many alterations, with the goal of creating one high-quality design. I propose to adapt this *conversational teaching model* for application in PCGML systems. While still having conversations with the tool about specific designs, the conversations are leveraged to talk about the general shape of the design space (rather than one specific output [147]).

[1]`https://github.com/mxgmn/WaveFunctionCollapse`

The artist trains the overall generative system to the point where it will be trusted to follow that style in the future, when the system can be run non-interactively. Instead of an individual artifact, the goal is to define a space of desirable artifacts from which the generator may sample.

It might seem most obvious to setup PCGML by using a generative statistical model. Such models explicitly capture the desired content distribution: $p(C)$. In the proposed discriminative learning strategy, I intend only to learn whether a candidate design is considered valid or not: $p(V|C)$. Such models can often be fit from much less data than generative models, and they do not require the distribution of content C in the samples to be representative of the target content distribution (easing the fundamental tension of PCGML). Indeed, many negative examples which demonstrate invalid designs will contain design details which should have zero likelihood. I assume the existence of a generator that can draw samples from a prior $p(C)$ filtered by the condition $p(V = true|C)$, effectively applying Bayes rule to draw samples from the conditional distribution: $P(C|V = true)$. WFC is one such generator that can be directly constrained to yield only designs that would be classified as valid.

This chapter illuminates the implicit use of machine learning in WFC, explains how discriminative learning may be integrated, and presents a detailed worked example of the conversational teaching model. I refer to the primary user as an artist to emphasize the primarily visual interface. It should be understood that the process of creating the input image can involve both design skills and programming reasoning: the artist is specifying both an aesthetic goal and a complex system of constraints to achieve that goal.

10.2 Background

In this section, I review WFC as an example-driven generator, characterize PCGML work to date as operating on only positive examples, and review the conversational interaction model used in mixed-initiative design tools.

10.2.1 WaveFunctionCollapse

WaveFunctionCollapse, as discussed in Chapter 9, is a content generation algorithm that primarily uses constraint solving but can be composed in many different ways depending on the implementation.[2]

In this chapter, I seek to extend the ideas of WFC while keeping them compatible with the existing implementations. One of the more unique aspects of WFC is that it is an example-based generator that can generalize from a single, small example image. In Sec. 10.6 I show that, while more than one example is needed to appropriately sculpt the design space, the additional examples can be even smaller than the original and can be created in response to generator behavior rather than collected in advance.

[2]As discussed by developers: e.g. @OskSta: "The generation algorithm is a spinoff of the Wave Function Collapse algorithm. It's quite content agnostic. I have a bunch of tweets about it if you scroll down my media tweets" `https://twitter.com/OskSta/status/931247511053979648`

10.2.2 PCGML

Summerville et al. define Procedural Content Generation via Machine Learning (PCGML) as the “generation of game content by models that have been trained on *existing* game content [emphasis added]” [344]. In contrast with search-based and solver-based approaches, which presume the user will provide an evaluation procedure or logical definition of appropriateness, PCGML uses a more artist-friendly framing that assumes concrete example artifacts as the primary inputs. PCGML techniques may well apply constructive, search, or solver-based techniques internally after interpreting training examples.

Machine learning needs training data, and one significant source of data for PCGML research is the Video Game Level Corpus (VGLC), which is a public dataset of game levels [345]. The VGLC was assembled to provide corpora for level generation research, similar to the assembled corpora in other fields such as Natural Language Processing. In contrast with datasets of game level appearance such as VGMaps,[3] content in the VGLC is annotated at a level suitable for constructing new, playable level designs (not just pictures of level designs).

The VGLC provides a valuable set of data sourced from iconic levels for culturally-impactful games (e.g. *Super Mario Bros* and *The Legend of Zelda*). It has been used for PCG research using autoencoders [170], generative adversarial networks (GANs) [367], long short-term memories (LSTMs) [343], multi-dimensional Markov chains [330, Sec. 3.3.1], and automated game design learning [265].

[3]`http://vgmaps.com/`

Some attempted solutions involve leveraging existing data. Snodgrass and Ontañon [332] train generative models but address the problem of small training data via transfer learning: training on data from related domains (level designs for other videogames). Sarkar and Cooper [303] similarly use data from adjacent domains to create novel designs via blending. However, Summerville et al. identify a "recurring problem of small datasets" [344]: most data only applies to a single game, and even with the efforts of the VGLC the amount of data available is small, particularly when compared to the more wildly successful machine learning projects. This is further complicated by our desire to produce useful content for novel games (for which no pre-existing data is available). Hence the fundamental tension in PCGML: asking an artist (or a team of artists) to produce quality training data at machine-learning scale could be much less efficient than just having the artists make the required content themselves.

Compounding this problem, a study by Snodgrass et al. [333] showed that the expressive volume of current PCGML systems did not expand much as the amount of training data increased. This suggests that the generative learning approach taken by these systems may not ever provide the required level of artist control. While this situation might be relieved by using higher-capacity models, the problem of the effort to produce the training data remains.

PCGML should be compared with other forms of example-based generation. Example-based generation long predates the recent deep learning approaches, particularly for texture synthesis. To take one early example, David Garber's 1981 dissertation proposed a two-dimensional, Markov chain, pixel-by-pixel texture synthesis approach [118]. Separate from Garber,[4] Alexei Efros and Thomas Leung contributed a two-dimensional, Markov-chain inspired synthesis method: as the synthesized image is placed pixel-by-pixel, the algorithm samples from similar local windows in the sample image and randomly chooses one, using the window's center pixel as the new value [99]. Although WFC-inventor Gumin experimented with continuous color-space techniques descending from these traditions, his use of discrete texture synthesis in WFC is directly inspired by Paul Merrell's discrete model synthesis and Paul Harrison's declarative texture synthesis [145]. Harrison's declarative texture synthesis exchanges the step-by-step procedure used in earlier texture synthesis methods for a declarative texture synthesis approach, patterned after declarative programming languages [150, Chap. 7]. Merrell's discrete 3D geometric model synthesis uses a catalog of possible assignments and expresses the synthesis as a constraint satisfaction problem [229]. Unlike later PCGML work, these example-based generation approaches only need a small number of examples, often just one. However, each of these approaches use only positive examples without any negative examples.

The strategy explored in this chapter—avoiding directly learning a generative model—is similar to the conceptual move in Generative Adversarial Networks (GANs). Rather than a training a directly generative model, GANs co-train distinct generator and discriminator models. GANs have been used for PCGML, for example in Mario-GAN [367], which uses unsupervised learning trained on levels from the Video Game Level Corpus (VGLC) [345] to generate levels based on the structure of *Super Mario*

[4]Efros and Leung later discovered Garber's previous work [98].

Bros.

Our work also uses example data to influence a distinct generator and discriminator, but these models are not represented by differentiable functions: they are instead represented by systems of constraints. Through a generate-and-test approach, learned validity constraints can be layered onto existing generative methods. For WFC, in particular, the validity constraints can be integrated into the core algorithm itself (a form of constraint solving).

Related to our approach, Guzdial et al. consider a use for discriminative learning in PCGML [148]. Rather than training a pattern-validity classifier (as in our work), they train a design pattern label classifier with labels such as "intro" and "staircase." While these labels do not reshape the space of (Mario level) designs the system will output, they allow the generator to annotate its outputs in designer-relevant terms.

10.2.3 Mixed-Initiative Design Tools

Several mixed-initiative design tools have integrated PCG systems. Their interaction pattern can be generalized as an iterative cycle where the generator produces a design and the artist responds by making a choice that contradicts the generator's last output. When the details of a design are under-constrained, most mixed-initiative design tools will allow the artist to re-sample alternative completions.

Tanagra is a platformer level design tool that uses reactive planning and constraint solving to ensure playability while providing rapid feedback to facilitate artist iteration [328]. Additionally, Tanagra maintains a higher-order understanding of the beats that shape the level's pacing, allowing the artist to directly specify the pacing and see the indirect result on the shape of the level being built.

The SketchaWorld modeling tool introduces a declarative "procedural sketching" approach "in order to enable designers of virtual worlds to concentrate on stating what they want to create, instead of describing how they should model it" [316]. The artist using SketchaWorld focuses on sketching high-level constructs with instant feedback about the effect the changes have on the virtual world being constructed. At the end of interaction, just one highly-detailed world results.

Similarly, artists interact with Sentient Sketchbook via map sketches, with the generator's results evaluated by metrics such as playability. As part of the interactive conversation with the artist, it also presents evolved map suggestions to the user, generated via novelty search [205, 208]. Although novelty search could be used to generate variations on the artist's favored design, it is not assumed that all of these variations would be considered safe for use. Tools based on interactive evolution do not learn a reusable content validity function, nor do they allow the artist to credit or blame a specific sub-structure of a content sample as the source of their fitness feedback. In Sec. 10.5, I demonstrate an interactive system that can do both of these.

As these examples demonstrate, mixed-initiative tools facilitate an interaction pattern: the artist sees a complete design produced by a generator and responds by providing feedback on it, which influences the next system-provided design. This two-way conversation enables the artist to make complex decisions about the desired outcome, without requiring them to directly master the technical domain knowledge that drives the implementation of the generator. The above examples demonstrate the promising potential of design tools that embrace this mode of control. However, they tend to focus on using generation to assist in creating specific, individual artifacts rather than the PCGML approach of modeling a design space or a statistical distribution over the space of possible content.

Despite the natural relationship between artist-supplied content and the ability of machine learning techniques to reflect on that content and expand it, PCGML-style generators that learn during mixed-initiative interaction have not been explored beyond the recent Morai-Maker-Engine [147]. In contemporary yerms, this discussion can be framed in terms of *human-centered machine learning* [300].[5]

10.3 Characterizing WFC as PCGML

Snodgrass describes WaveFunctionCollapse as "an example of a machine learning-based PCG approach that does not require a deep understanding of how the algorithm functions in order to be used effectively" [330, Sec. 2.8]. WFC learns a generalized vocabulary that opens up space for exploring alternative learning strategies. The following description dives into the what and how of that generalized vocabulary.

Rather than being a single monolithic input-to-output-mapping function, WaveFunctionCollapse is a pipeline of multiple stages. While most prior discussion has emphasized the constraint-solving stage, the pattern learning steps that precede it are

[5]Such as the focus of the Human-Centered Machine Learning Perspectives Workshop `https://gonzoramos.github.io/hcmlperspectives/`

equally important. It starts with an analysis of an image to a vocabulary of valid adjacencies between patterns. In the second phase, the results of the analysis are used as constraints in a generation process identifiable as constraint solving [186].

In this section, I describe how the pattern learning steps are an instance of machine learning. In particular, I show that this phase learns three functions: which pattern is present at each location (the pattern classifier); if the pattern adjacency pairing is within the artist's preferred style (the adjacency validity); and how a location in the output should be rendered (the pattern renderer) given the constraint-solving generator's choice of pattern placement assignments. Fig. 10.1 illustrates the relationship between these three functions.

10.3.1 Tiles

Conventionally, the space WFC operates on is a rectangular grid. This gives us an easy way to define the nodes and adjacency edges: the nodes are each intersection on the Go board, and the edges are the lines between them. However, WFC can work on any graph for which a well-defined adjacency function can be specified. For example, Stålberg's experiments with triangular meshes.[6] These can be non-spatial: to create a looping animation Matt Rix added a time edge.[7] Similarly, Martin O'Leary's poetry generator[8] uses non-neighbor edges for rhyming patterns and scansion.

Multi-node tiles are also possible: one of the *Bad North* [340] innovations was

[6]`https://twitter.com/OskSta/status/784847588893814785`
[7]`https://twitter.com/MattRix/status/872674537799913472`
[8]`https://github.com/mewo2/oisin`

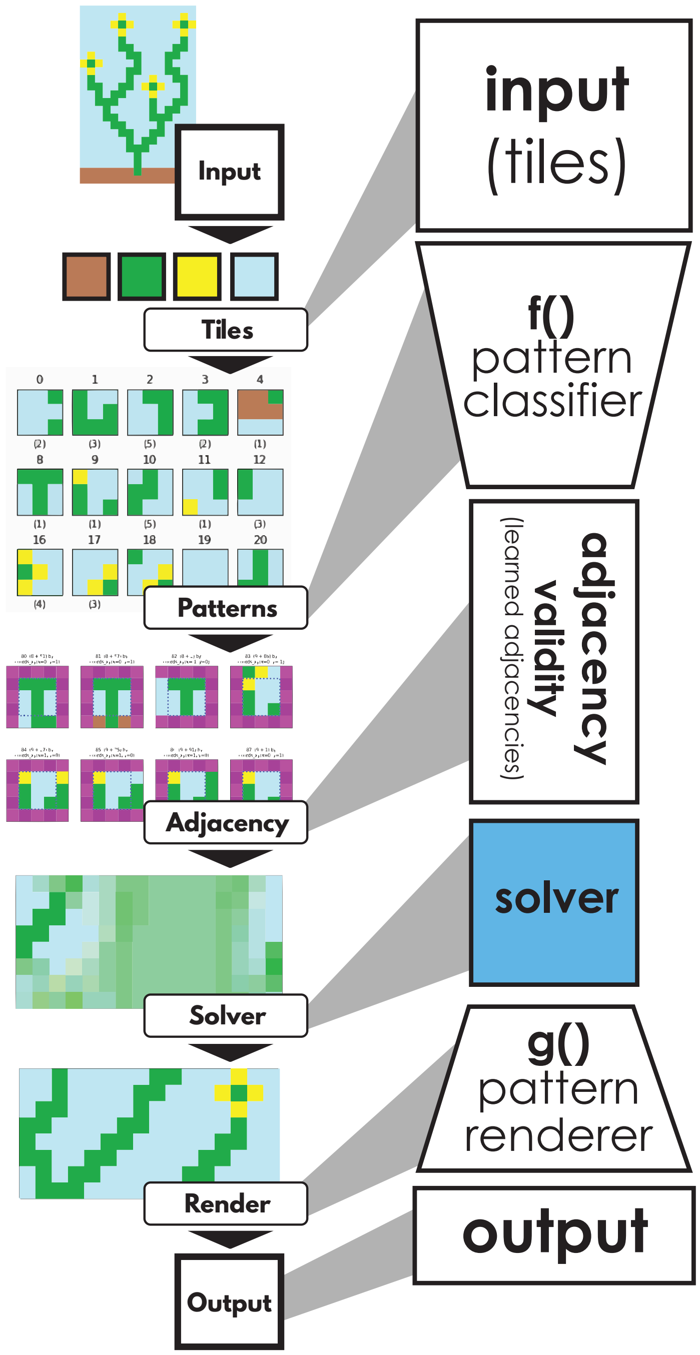


Figure 10.1: WaveFunctionCollapse pipeline. The left column shows the pipeline steps. The right column indicates how the steps match the functions learned from pattern analysis.

to include larger elements as possible modules.[9]

WaveFunctionCollapse works by placing small elements into the context of a larger whole. While these puzzle pieces go by many different names in the wild, I refer to them as tiles. A tile can be a single pixel, a 2D image, 3D geometry, a word, or any other distinct modular component.

Typically, tiles are either specified by an artist or extracted from the training data. The solver does not care about the contents of the tiles, only about the adjacencies between tiles.

10.3.2 Pattern Classifier

The heart of WFC are the *adjacencies* that describe the constraints between patterns, used by the constraint solver to generate the solution. Gumin's original implementation of WFC has ways of defining patterns: a `SimpleTileModel` that specifies adjacencies between individual tiles by hand and an `OverlappingModel` that infers the relationships between tiles in their local contexts. Other models are possible: for example, *Bad North* [340] learns which modules can be adjacent by comparing the vertices at the edges of the 3D model and looking for matching profiles in the shared plane.

With this learning step, the `OverlappingModel` operates on what it refers to as *patterns*. A pattern is a tile plus the context of its surrounding adjacencies, as found in the training data. These are usually $N \times N$ regions of tiles (where N is typically 2 or 3, as larger values need more training data and computing resources). The classifier reduces

[9]A conceptually simpler way to implement multi-tile elements is to give the different parts the constraint that the only allowed neighbor in the relevant edge is another part of the multi-tile module.

pertinent details about the surrounding context into a single value. By operating on patterns rather than directly on tiles, the solver can make use of the implied higher-order relationships that are learned from the training data.

The `OverlappingModel` also uses reflection and symmetry to augment the training data. Readers with a machine learning background will recognize this as a *data augmentation* strategy [53, pp. 138–142]. This can be configured for the training since some input images, such as a flower viewed from the side, have a strong directionality.

In Gumin's implementation of WaveFunctionCollapse, the pattern classifier is a relatively simple 1:1 mapping of patterns learned from the training data. Other learning methods are possible: for example, grouping semantically similar tiles via k-means. Or we can imagine a deep convolutional neural network being used to map as-yet unseen tile configurations into the existing pattern catalog, so long as they were perceptually similar enough.

Pattern classification need not be a strictly local operation. If we wanted to generate dungeon levels for a roguelike game, we may be particularly interested to note which treasure chests are easily reachable by the player versus not. Or, in platformer level generation, this could distinguish rewards placed on the player's default path (as a guide) or off the path (as an enticement to explore). In the future, I imagine the contextual information used in the pattern classifier to come from many different sources. In the *texture-by-numbers* application of image analogies [155], the artist hand-paints an additional input image to guide the interpretation of the source image and the generation of the target image in another example-driven image generator. In Snodgrass' hierarchical approach to tile-based map generation [331], a lower-resolution context map is generated automatically using clustering of tile patterns.

10.3.3 Patterns and Adjacency

Gumin's WFC operates not on the tile constraints, but rather on the constraints between patterns (Fig. 10.2). This is a generalization of the adjacencies between individual tiles: the pattern classifier captures additional adjacency information about the local space, similar to an image filter kernel or the convolutions used in image processing and CNNs. In effect, each tile is treated as the tile-plus-its-context: we prefer some adjacencies, such as placing a flower in a flowerpot. Other adjacencies are non-preferred: the flower should not be growing in the middle of a carpet. In Gumin's WFC implementation, adjacencies are stored as a (usually sparse) multidimensional matrix, with dimensions of $patterns \times patterns \times directions$.[10]

We can characterize the method used to learn the adjacency legality as Most General Generalization (MGG), the inverse of classic Least General Generalization (LGG) inductive inference technique [274]. Gumin's implementation simply allows any tile-compatible overlapping patterns to be placed adjacent to one another, even if they were never seen adjacent in the single source image. A side effect of this is that any pattern adjacencies seen in the source image (which are tile-compatible by construction) must be considered valid for the generator to use later. While MGG might appear as

[10]Constraint solvers that are not optimized for grids of constraints can, instead, use a set of the allowed adjacencies to specify the constraints. This list of tuples is isomorphic with the pattern matrix and is more convenient for adding or subtracting adjacencies from the allowed set.

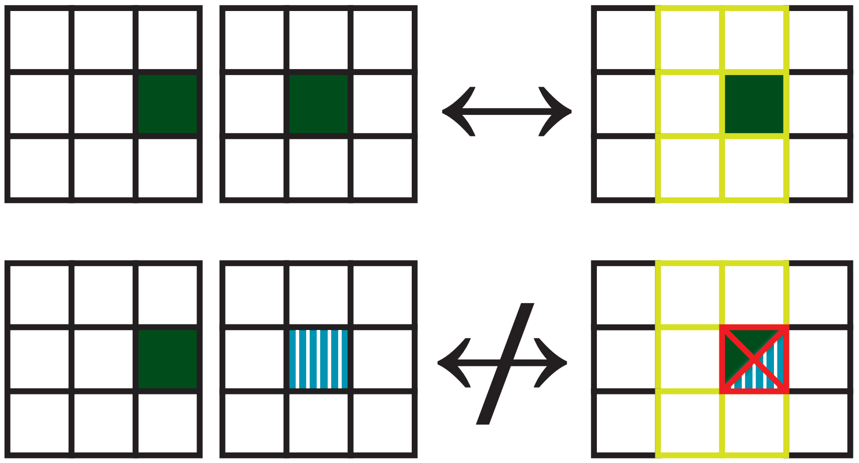


Figure 10.2: An example of a pattern overlap, with a [0,1] offset. The top pair of patterns is a legal overlap, because of the valid intersection. The bottom pair is not a legal overlap, because the striped blue tile and the solid green tile conflict.

simple parsing and tallying, something too simple to be considered as machine learning, it is useful to compare this approach with other classic machine learning techniques like Naive Bayes [299, Chap. 20]. Naive Bayes classifiers are trained with no more sophistication than tallying how often each feature was associated with each class.

10.3.4 Using Multiple Sources of Training Data

The art of constructing the single source image for Gumin's WFC often involves some careful design to include all of the patterns that are preferred and none that are non-preferred. By allowing for multiple positive and negative examples and using a slightly altered learning strategy, I show how this meticulous work can be replaced with a conversation that elaborates on past examples.

While many WFC implementations use a single image for training data, this is an interface detail rather than an intrinsic limitation of the algorithm. Using multiple images allows for discontinuities in the training data, which simplifies the expression of some complex relationships. Equally, since the adjacencies between patterns are just a set of tuples[11] we can also use *negative examples* (at the cost of increased interface complexity) that remove adjacencies from the allowed set.

One of the reasons that WFC was rapidly adopted was that artists could create complex constraints by painting a picture. Complicating the interface removes some of this advantage. However, other equally approachable ways to specify constraints have already been explored—for example, *Bad North* [340] automatically detects alignment between the 3D geometry of neighboring tiles.

Gumin's pattern classifier function implicitly captures the relationships between patterns in the training data. The first and most absolute distinction is between legal and illegal overlaps. Because the patterns in the `OverlappingModel` need to be able to be placed on top of each other without contradictions, some patterns will never be legal neighbors. If one 3×3 pattern has a blue center tile, while another 3×3 pattern has a green right tile, the green-right-tile pattern can never be legally placed to the left of the blue-center-tile pattern (Fig. 10.2).

Gumin's MGG learning strategy is hardly the only option possible, even using the default pattern classifier. A LGG learning strategy would only allow those adjacencies explicitly demonstrated in the source image. However, this highly-constrained

[11]or a sparse matrix

alternative might not allow any new output to be constructed that was not an exact copy of the source image. Likely, the ideal amount of generalization falls somewhere between these extremes.

In a discriminative learning setup, we might consider all adjacencies explicitly demonstrated in positive example images to therefore be positive examples for the learned adjacency relation. Likewise, a negative example image needs to demonstrate at least one adjacency that would be considered invalid by the learned relation. I refer to the artist's intended set of allowed relations as the *preferred set.* Artists can attempt to adjust the legal set to match the preferred set, but this requires a combination of technical reasoning and trial-and-error iteration.

Gumin's MGG strategy does not try to generalize from the observed patterns or to infer non-observed but possibly still preferred adjacencies. Artists can attempt to adjust the legal set to match the preferred set, but this requires a combination of technical reasoning and trial-and-error iteration. Or, alternately, they could switch to using the `SimpleTiled` model for which they directly (and with considerable tedium) specify the complete set of allowed adjacencies.

This is a limitation of the learning strategy, but not the WFC algorithm itself. The output of the `agrees()` validity function in Gumin's implementation just checks if two patterns can legally overlap, but any arbitrary adjacency validity function can be substituted here. As long as the validity function can be computed over all pairs of patterns, it can act as the whitelist for the constraint domains without changing the WFC solver itself.

10.3.5 Additional Pre-Solving Constraints

While most uses of WFC to date have encoded all information about the long-range constraints in the training image data, an implementation with a more specific application in mind has the opportunity to include additional constraints. For example, including a reachability constraint in a level generator (so that there is a path to all the rooms in the level) can be implemented as a global constraint.

The most common use of additional constraints in the wild is to pre-seed the solver with partial solutions. For example, the `Flowers` example is pre-seeded with the lowest row limited to patterns that include the brown soil pixels. Similarly, the *Caves of Qud* level generator uses WFC as part of a pipeline, with additional details added in the empty spaces between the walls that WFC adds. One useful side-effect of this is that WFC can complete partial solutions. When used as part of a mixed-initiative generator, this means that the user can draw a partial solution (such as the main path through a level) and have the generator fill in the rest of the space with relevant and contextual content.

10.3.6 Solver

The adjacency data is sent to the constraint solver. Only the constraint data itself is needed: the list of constraints is sufficient. The constraint solver can be implemented as: Gumin's stochastic observation/propagation solver; an ASP solver like Clingo;[12] a solver that uses a constraint modeling language, such as MiniZinc;[13] and

[12]`https://github.com/potassco/clingo`
[13]`https://www.minizinc.org/`

so on. Though it is not the focus of the present chapter, the properties of the solver can vary. Most commonly, this takes the form of different heuristics or the addition of backtracking, which is most applicable for tile sets that greatly depart from the properties of the original examples (small N, adjacencies that are nether too constrained or too open). With Gumin's parameters, the long-range constraint propagation in WFC is more important for its high success rate without backtracking, rather than the heuristic used [186].

10.3.7 Rendering

The final step of WFC is an inversion of the pattern classification: translating the grid of selected patterns back into a grid of tiles (and tiles into pixels). Interesting animations showing the progress of generation in WFC result from blending the results of the pattern renderer for all patterns that might yet still be placed at a location. Animations of these visualizations over time attracted several technical artists (and the present authors) to learn more about WFC.

Generalizing the role of the pattern classifier, we can imagine other functions which decide how to represent a local patch of pattern placements. Again, we can imagine the use of a deep convolutional neural network (CNN) to map a small grid of pattern identifier integers into a rich display in the output. Although the pattern renderer's input datatype is fixed, the output can be whatever artist-visible datatype was used as input to the pattern classifier (whether that be image pixels, game object identifiers, or a parameter vector for a downstream content generator).

If additional annotation layers are used in the source images it is reasonable to expect that the output of the generator could also have these annotation layers. (As in the texture-by-numbers application mentioned above, the navigability criteria in *Bad North*,[14] or the player path data present in some VGLC data.) For platformer level generation, the system could output not only a tile-based map design, but also a representation of which parts of the map player can actually reach. Likewise, if the tiles are more complex than static images, the rendering function can output things other than the final image. For example, a level generator might have each tile represent a *room generator* rather than the final level geometry, and the rendering output would be the parameter data for the calls to the room generators.

10.3.8 Variation in Implementations

Each of the above steps can be replaced. The many implementations of WFC in the wild frequently vary the features that each step uses: for example, *Bad North* doesn't use patterns to learn tile adjacency, and instead relies on matching geometric profiles of 3D models.

To prototype a variation of WFC supporting an alternate pattern classifier, pattern renderer, and adjacency validity function, I initially developed a surrogate implementation of WFC in the MiniZinc constraint programming language. Later, I integrated the specific ability to generate with a customized pattern adjacency whitelist into a direct Python-language clone of Gumin's original WFC algorithm.

[14]`https://twitter.com/OskSta/status/917405214638006273`

10.4 Vivisecting WFC

Viewed compositionally, WFC is composed of several distinct and modular parts (Fig. 9.2). When using the overlap model, the first step is to deconstruct the input image from tiles into patterns: a transformation operation. The learning step generates an annotation for those patterns: the adjacency table. The adjacency table is interpreted into the rules the constraint solver follows. The solver uses the constraint rules and generates a new solution, which is transformed back into tiles.

By looking at the individual components, we can identify the operations in a generative system that can be swapped or modified to enable new behaviors. The adjacency table, for example, can be computed online or offline. If we want to reuse the same constraints we can calculate them once and cache the result. However, if we want to change the rules dynamically, particularly in a mixed-initiative system, we can get more creative with our interventions.

10.5 Setting Up Discriminative Learning

As discussed above, the original approach in WFC is to define the adjacency validity function with the most permissive possible way, using every possible legal adjacency as the valid set. Among other drawbacks, this requires careful curation of the patterns so that every adjacency in the legal set is acceptable. While allowing very expressive results from a single very small source image, there are many preferred sets that are difficult to express in this manner. However, this is just one of the many possible strategies. An anomaly-detection strategy, such as a one-class Support Vector Machine [253], might allow the set of valid adjacencies to more closely approximate the ideal preferred set, allowing the artist to use patterns with a much larger legal adjacency set.

In this section, I consider the presence of possible negative examples. By removing adjacency pairs that the artist explicitly flagged as undesirable, we can more precisely determine the valid set. By default, WFC uses all of the legal adjacencies, but the preferred-valid set can be adjusted to include more or less of the legal set, with corresponding effects for the generation.

10.5.1 Machine Learning Setup

In addition to the single positive source image used by the original WFC, I introduce the possibility of using more source images. Some of these are additional positive examples: it can be easier to express new adjacencies while avoiding unwanted ones by adding a completely separate positive example. In the positive examples, every adjacency is considered to be valid, as usual. In contrast, sometimes it is easier to specify just the negative adjacencies to be removed from the preferred set. Note that these additional example images do not need to be equal in size. In fact, a tiny negative example, showing just the undesirable pairing, lets an artist carve out one bad location in an otherwise satisfactory design (cf. Fig. 10.3).

Finally, we have the validity function: a function that takes two patterns (plus how they are related in space, e.g. up/down/left/right) and outputs a Boolean evaluation of whether or not their adjacency is valid. In the original WFC, this is simply an overlapping test: given this offset, are there any conflicts in the intersection of the two patterns? However, as suggested above, there are more sophisticated validity functions that also produce viable results.

10.5.2 Human Artist Setup

In our mixed-initiative training approach, we expect the artist to provide at least one positive example to start the process. The image should demonstrate the local tiles that might be used by the generator, but it does not need to demonstrate all preferred-valid adjacencies. Note that providing one example is the typical workflow for WFC (unmodified, Gumin's code only accepts one example). However, instead of expecting the artist to continue to iterate by changing this one example, which can quickly grow complex, the artist can isolate each individual contribution.

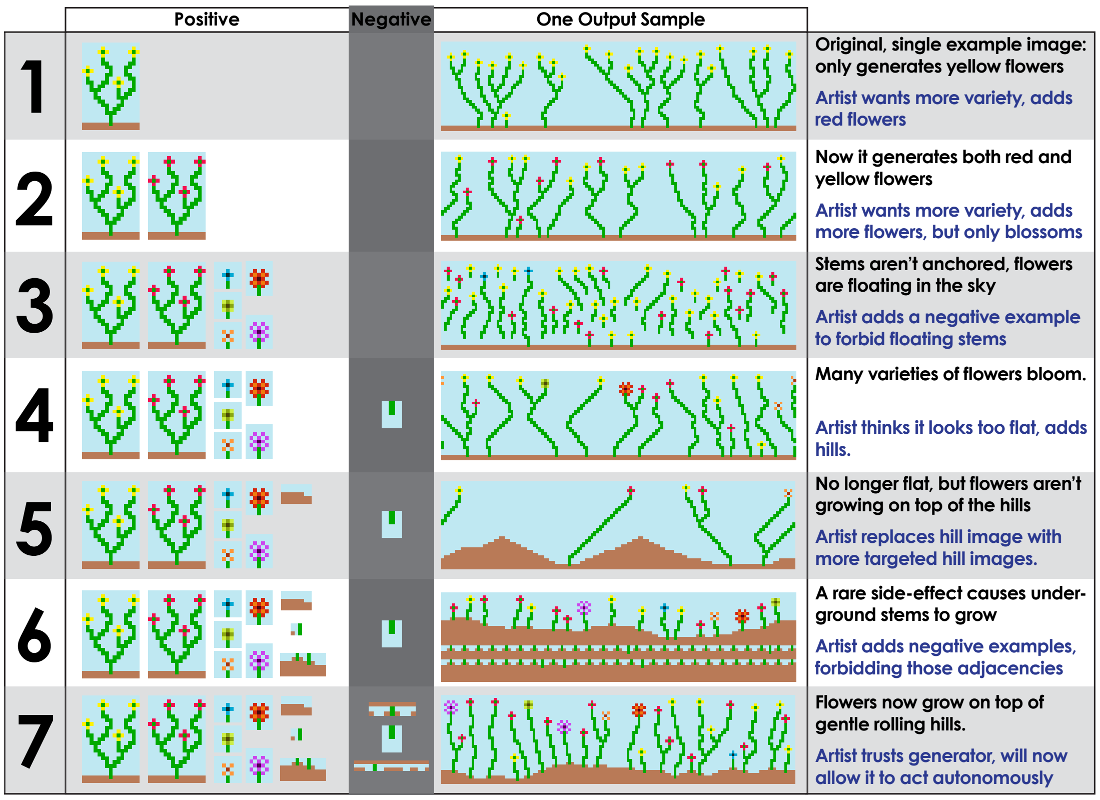
Initially, we set the whitelist of valid adjacencies to be fully permissive (MGG), covering the legal adjacencies of the known patterns. From this, we generate a small portfolio of outputs to sample the current design space of the generator. Even a single work sample is often enough to spur the next round of interaction. The artist reviews the portfolio to find problems. They can directly add one or more generated outputs to the negative example set, crop an example to make a more focused negative example, or hand-create a clarifying example. If additional positive examples are desired to increase variety, those can also be added—although they may immediately prompt the need for negative examples to address over-generalization.

With the new batch of source images, in a trivial amount of time we retrain each of the learned functions: the pattern classifier, the adjacency validity function, and the pattern renderer. The update pattern classifier defines the space of patterns that might be placed by the generator. The newly-learned validity function defines the updated whitelist used in the constraints. The updated pattern renderer might even display existing pattern grids in a new way. As before, we sample a portfolio. The artist repeats the process until they are satisfied with the work samples. The result is a generative system with a design space that has been sculpted to the artist's requirements, all without the artist needing to understand or alter any unfamiliar machine learning algorithms.

10.6 Worked Example

In this section we will walk through an example run of the conversational interaction an artist has with WFC when using a discriminative learning setup. The conversation takes place over several iterations that are visually represented in Fig. 10.3. All outputs shown were generated by executing a minimally modified WFC implementation. The alterations to the implementation are adding patterns from multiple images, paired with removing items from the adjacency whitelist. The resulting pattern grids are rendered with the pattern renderer.

In this running example, we will make use of a refinement to the MGG strategy used in Gumin's implementation. Rather than simply allowing all patterns which agree



1 Original, single example image: only generates yellow flowers
Artist wants more variety, adds red flowers

2 Now it generates both red and yellow flowers
Artist wants more variety, adds more flowers, but only blossoms

3 Stems aren't anchored, flowers are floating in the sky
Artist adds a negative example to forbid floating stems

4 Many varieties of flowers bloom.
Artist thinks it looks too flat, adds hills.

5 No longer flat, but flowers aren't growing on top of the hills
Artist replaces hill image with more targeted hill images.

6 A rare side-effect causes underground stems to grow
Artist adds negative examples, forbidding those adjacencies

7 Flowers now grow on top of gentle rolling hills.
Artist trusts generator, will now allow it to act autonomously

Figure 10.3: A worked example of the mixed-initiative conversational teaching model process. The artist observes the results of each step (top black text) and makes a change for the next step (lower blue text). Each step adds either positive or negative examples. The source images for each output can be seen in the columns on the left, with a representative output image to the right. Note the positive patterns being added wrap around: the floating stems could also be suppressed by not adding the positive stem-above-air pattern, but that would result in the exact same constraint rules as expressing it via the negative pattern.

on their overlapping tiles, we allow all such patterns *except* those taken from negative examples. Indeed, this is still the most general generalization possible under the extra constraints. Working through the conversation, our artist begins Iteration 1 with the algorithm by supplying a single positive example. Here we use the `Flowers` example taken from Gumin's public repository. MGG learns the legality relations exactly like the original WFC. Using the generator to produce work sample, the artist decides that the image needs more colorful flowers.

In Iteration 2, the artist augments the positive image set with a second image, having repainted the flowers to be red. Adding additional patterns to Gumin's WFC required only minimal code changes. In the resulting work sample, now both red and yellow flowers are seen (new patterns were made available to the generator). However, the artist still wants more flower variety.

Rather than copy-pasting the original tiles again, this time the artist creates a number of smaller samples that focus exactly on what they want to add to the composition. These extra tiny examples might throw off statistics in a generatively trained model. In the work sample for Iteration 3, the new flowers are present but a surprising new phenomenon arises. This possibility of floating stems results from the particulars of WFC's default pattern classifier and adjacency relation function learning. The artist is not concerned with these implementation details and wishes simply to fix the problem with additional examples.

By selecting and cropping a 3×4 region of the last work sample, the artist creates the focused negative example for use in Iteration 4. MGG, now with the extra constraint from the negative example, no longer considers floating stems to be a possibility—despite the fact that this pattern can be identified in the input by the pattern classifier. The work sample is now free from obvious flaws.

Having previously only considered very small training examples, the artist notes a particular feature of the larger generated output. The ground appears uninterestingly flat. In Iteration 5, the artist provides a small positive example of sloped hills, hoping the generator will invent a rolling landscape. However, the work sample for this iteration suggests that the generator has not picked up the generality of the idea from the single tiny example—it only knows how to build continuous ramps without any flowers.

In Iteration 6, a few more positive examples show that stems can be placed on hills and that the bumps of hills can be isolated (not always part of larger ramps). However, in rare circumstances the generator will now place stems underground. This would not have been spotted without examining many possible outputs, highlighting the importance of a tool that allows the artist to give feedback on more than one example output.

Finally, in Iteration 7, the artist is able to get the look they preferred. Adding negative examples to take care of the edge cases is easy and can be done without adjusting the earlier source images. Testing shows that the generator is reliably producing usable images. Because of the iterations, the artist now has enough trust in the generator to allow it to perform future generation tasks without supervision. The learned pattern classifier function, pattern renderer function, and adjacency legality function compactly summarize the learning from the interaction with the artist.

In this worked example, every new training example added beyond the first is a direct response to something observed in (or observed to be missing from) concrete images produced by the previous generator. Many demonstrate patterns that are not what the generator should produce in the future, even if it is not realized that this was the case earlier. Instead of iterating to produce a carefully curated set of 100% valid examples, we make progress by adding focused clarifications.

10.7 Conclusion

The fundamental tension in PCGML is that the effort to craft enough training data for effective machine learning might undermine the motivation to use PCGML in the first place. This makes many machine learning approaches impractical: even when the design goal is flexibility (rather than nominally infinite content) the immense amount of training data required can be daunting.

However, existing approaches to single-example PCG, such as WaveFunction-Collapse, suggests that small-training-data generators are possible. By looking at WFC from a compositional standpoint, we can identify where we can alter the generative system to incorporate a mixed-initiative process.

Augmenting our WFC implementation with a discriminative learning strategy, we can leverage the usefulness of focused negative examples, or even just example fragments. As our worked example of the conversational teaching model shows, an artist can intuitively make targeted changes without being overly concerned about maintaining a representative distribution or disturbing earlier, carefully planned patterns just to fix a rare edge case.

Combining PCGML with mixed-initiative design assistance tools can enable artists to sculpt a generator's design space. Rather than building just one high-quality artifact, the artist can train a generator through iterative steps to the point where they trust it for autonomous generation.

10.8 Acknowledgments

This chapter is based on work originally published as "Addressing the fundamental tension of PCGML with discriminative learning" [187] which was co-authored with my advisor, Adam M. Smith. I happen to be particularly fond of figure 10.3.

The authors wish to thank Adam Summerville for the extensive feedback which greatly improved this research.

Chapter 11

Neurosymbolic WFC

11.1 Introduction

The previous chapter demonstrates that a compositional change to WFC can have a significant effect, enabling a new interaction approach. (RQ3) However, we can go further. WFC is generally classified as a constraint solving algorithm. As we've seen in the previous chapters, from a compositional standpoint it is somewhat more complicated. Knowing this, can we perform further interventions? For example, can we take this purportedly constraint-solver-based system and combine it with an algorithms representative of sophisticated, contemporary machine learning? This chapter offers a variation on WFC that combines the symbolic AI method of constraint solving with a neural machine learning method.

Image generation systems based on *neural* networks have been able to create convincing, high-fidelity images, from faces [178] to animals [43]. This capability has

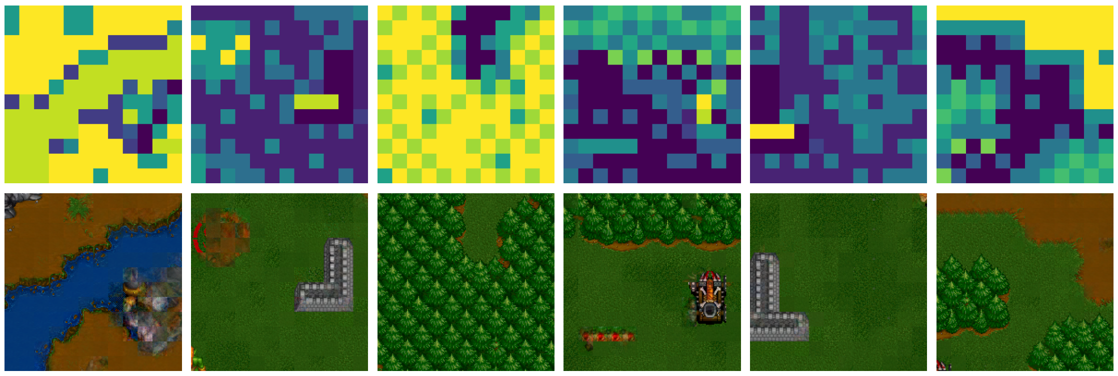


Figure 11.1: Unconstrained WFC + Neural Decoder outputs. Note the directional transitions between terrain types, units spanning multiple tiles, and an imperfecty reconstructed building (only one instance of this building is seen in the training input). Top visualizes the latent tile selected at each location (from a vocabulary of just 12 latent codes) while bottom shows synthesized color images.

been of interest to the game content generation community [209]. However, one common shortcoming of these learning-based methods is that they are difficult to control. Simply asking designers to provide more and more examples of the desired kind of content is not a satisfying alternative [186]. Controllability is especially important when it comes to generating videogame maps. In order to support specific player experiences, a generated map must obey hard constraints [320]. *Symbolic* artificial intelligence (AI) methods, such as constraint-solving, offer the ability to directly enforce key properties on the output of a generator [321], but it is challenging to combine them with neural techniques.

In this chapter, I show a *neurosymbolic* [119] approach to generating videogame map images in which discrete representation learning methods (VQ-VAEs) [361] are used to produce a vocabulary of latent tile descriptors. Novel arrangements of tiles produced by a constraint-based map generator (WFC) [186] are rendered in a context-sensitive way, allowing the expression of a large effective tileset despite working with a small

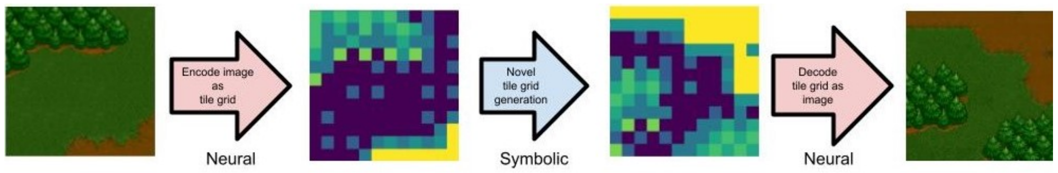

Encode image as tile grid
Neural
Novel tile grid generation
Symbolic
Decode tile grid as image
Neural

Figure 11.2: Overview of our neurosymbolic map generation approach. Grid generation can accept additional constraints on the tile grid.

latent vocabulary. Figure 11.1 demonstrates the results of applying our method to a *WarCraft II* map setting.

11.2 Approach: VQ-VAE + WFC

Using the WaveFunctionCollapse (WFC) algorithm (a family of symbolic constraint-solving methods) to generate a new map image requires a carefully curated tileset. The input map is described using indexes into that tileset and, after generating a new map of tile indexes, they can be replaced with the corresponding tile image to create a map image. Our method (sketched in Figure 11.2) obviates the need for a tileset by learning tiles using a vector-quantizing variational autoencoder (VQ-VAE). The learned VQ-VAE model provides both a mapping from patches of the original image to the corresponding tile indices, as well as a (context-sensitive) mapping from the tile index grid back to image pixels.

To be more specific, the method works as follows:

1. The neural encoder (a sub-network of the VQ-VAE) is used to reduce a training map image into a set of discrete latent tile indexes, creating a tile grid.

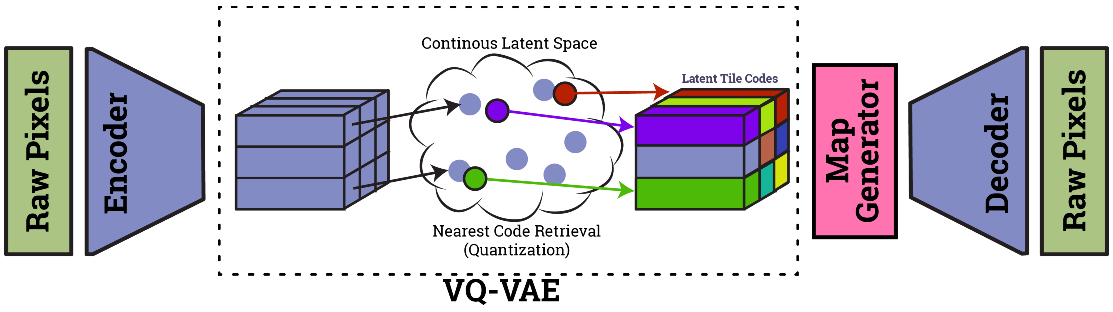


Figure 11.3: Overview of our VQ-VAE used to compress continuous latent vectors into a discrete latent tile code vocabulary. The latent tile codes are then used by the WaveFunctionCollapse algorithm to generate novel maps.

2. The symbols constraint-solving algorithm (WFC) uses this grid to infer which tiles can go next to another, and it uses these adjacency constraints to generate a new tile grid (under optional additional user-specified constraints), representing a novel map.

3. The neural decoder is then used to render the tile grid back into the original pixel granularity in a context sensitive way (the appearance of a single tile can be influenced by the selection of tiles placed next to it in the grid).

A traditional autoencoder (AE) learns to summarize input images using a bottleneck representation consisting of a vector of continuous values. A vector-quantizing (VQ) autoencoder, on the other hand, uses discrete integers as the bottleneck representation. A trainable codebook supplies the vectors used in the decoder, and encoded vectors are compressed by assigning them the integer index into the codebook of the vector most similar to them by Euclidean distance.

In the training phase for our VQ-VAE model (shown in Figure 11.3), large patches of the training design image are passed through a convolutional encoder model that reduces the image into a low-resolution grid of high-dimensional vectors. Each vector on this grid is used to lookup the nearest codebook vector. These quantized vectors are assembled into a grid with the same shape as the input to the quantization process. After this, a convolutional decoder model transforms the grid into a high-resolution color image. The loss function forces this to resemble the original input. For more detail on the use of VQ-VAE to learn tiles for image generation, the reader should consult the upstream machine learning literature [361].

Once training is complete, we can use the learned (quantizing) encoder to convert the large training design image into a tile index grid using the latent tile vocabulary. This grid is used by WaveFunctionCollapse to generate novel tile index grids. WFC consists of two parts: the adjacency learning, which translates the training data into constraint rules, and the constraint solver, which generates a new configuration of rules that satisfies the constraints. For adjacency learning, we used standard overlapping WFC with 2×2 tile patterns, which defines the constraint rules as patterns of tile neighborhoods [186]. The solver then finds a solution that satisfies the rules, generating a new grid of tile indices. Using the trained (dequantizing) decoder,[1] we can synthesize novel high-resolution pixel grids (Fig. 11.6).

[1]The *decoder* in an autoencoder works analogously to the *generator* in GAN models.

11.3 Demonstrations

To show how this method can generate novel levels, we apply it to level images of two tile-based games: *Warcraft II* and *Super Metroid*. A separate VQ-VAE model (encoder + decoder) is trained for each map.

11.3.1 WarCraft II

Warcraft II is a real-time strategy game released in 1995 by Blizzard Entertainment. Images of Warcraft II levels[2] were used to demonstrate VQ-VAE + WFC. Figure 11.4 shows the intermediate (quantized) latent tile representation of the training design image and the corresponding output of the trained VQ-VAE model in reconstructing that data. Although some of the textured detail of the tiles is missing, the overall image is quite similar to the input. Note how the light blue "land" tile next to the dark blue "shore" tile successfully renders all needed cases of a coastline.

Figure 11.1 shows some results of our approach. Contextualized rendering allows many important effects. The different terrain types blend together with transitions that depend on the surrounding tiles, the direction of the border, and the neighboring terrain types. Because of this, many different tile images can be represented with only 12 latent tiles.[3] Consider the two yellow tiles in Figure 11.4. Even though they are represented by the same index, they are reconstructed into different pixel images–one

[2] `https://vgmaps.com/Atlas/PC/WarCraftII-BeyondTheDarkPortal-Humans-Mission01-Alleria'sJourney.png`

[3] A 3×3 grid with 12 possible placement in each location gives rise to billions of possible combinations, some of which are directly supervised by the training data and others to be covered by the decoder network's generalization abilities.

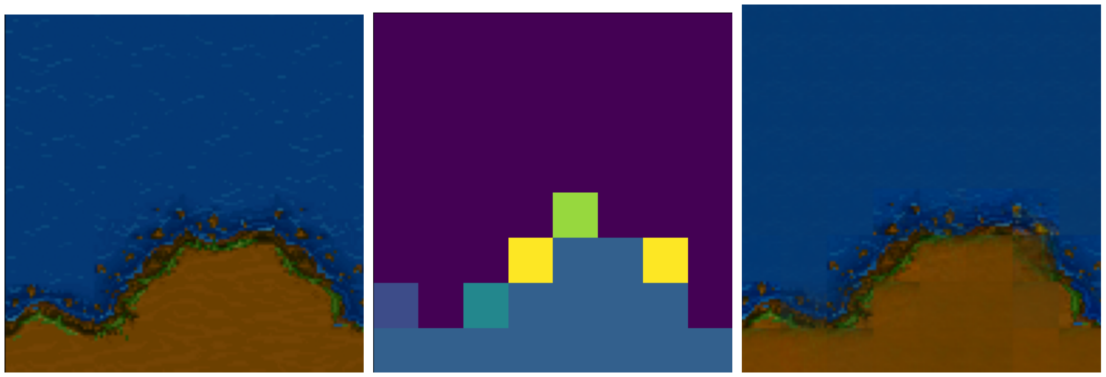


Figure 11.4: The autoencoder model reconstructs colored images after passing them through a bottleneck of discrete latent codes. Left, original data from WarCraft II map. Middle, the discrete latent variables in the bottleneck layer of the VQ-VAE. Right, the reconstructed image from the latent variables.

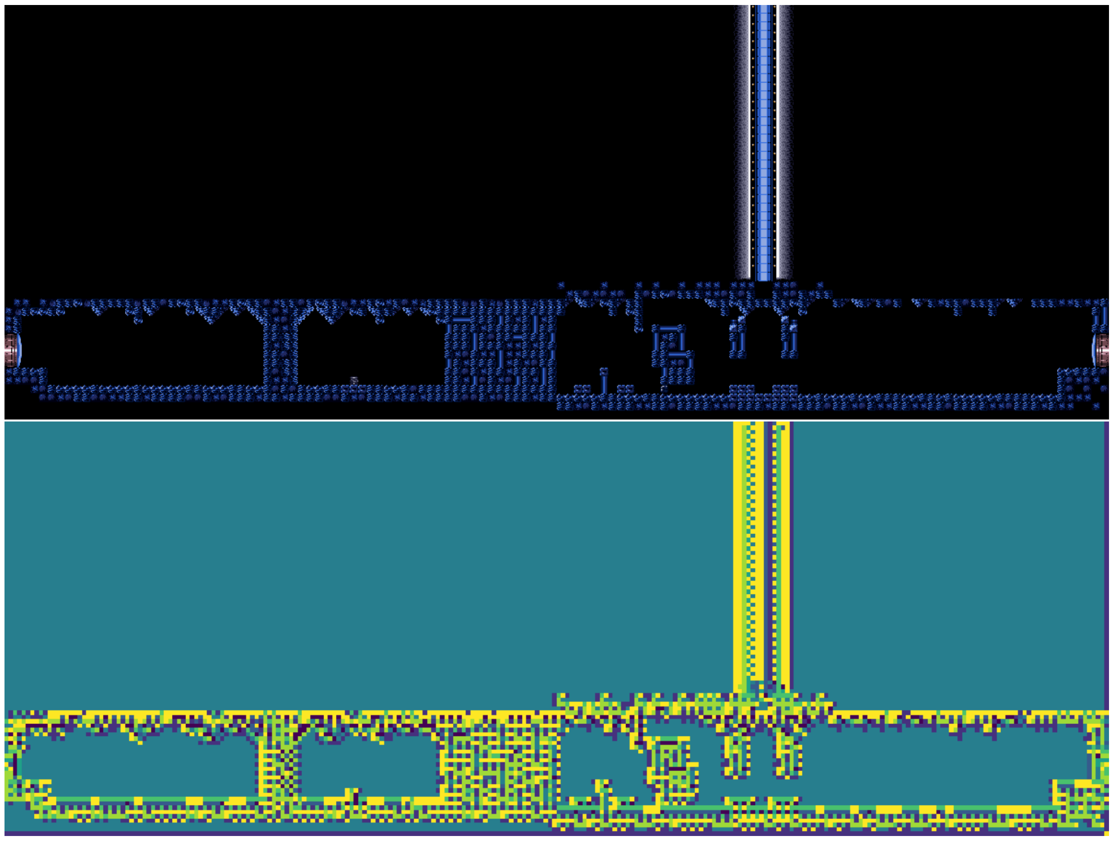


Figure 11.5: Selected segment of the *Super Metroid* map. Left shows source pixels while right shows assigned latent codes (8 latent codes).

for a “left facing coast” tile and the other a “right facing coast” tile. This capability allows us to reconstruct maps that respect tile transitions and unit placements with a comparatively low number of distinct tile codes.

11.3.2 Super Metroid

Super Metroid is a platformer released in 1994 by Nintendo. As with *Warcraft II*, we use a single level image to train the VQ-VAE network, and the resulting encoded image is used to create the tile sets for WaveFunctionCollapse (see Figure 11.5). Fewer latent codes were used in Metroid (8 vs. 12), since the in-game image representation of any given tile does not depend on neighboring tiles. Figure 11.6 shows novel level images generated using WaveFunctionCollapse.

However, unlike the *Warcraft II* examples, these level images were generated under specific constraints that certain tiles be part of a door or an item pedestal. Given a partial grid of latent tile codes, WFC can correctly complete the blank regions. To control the output (placing a door or an item in a certain location), a user can first use the encoder on the original input to find the arrangement of latent tiles for a given feature. Then, the user can provide a partial grid, with those latent tiles at the desired position. These constraints influence the latent tile grid generated by WFC, without changing the way that the renderer creates the level image.

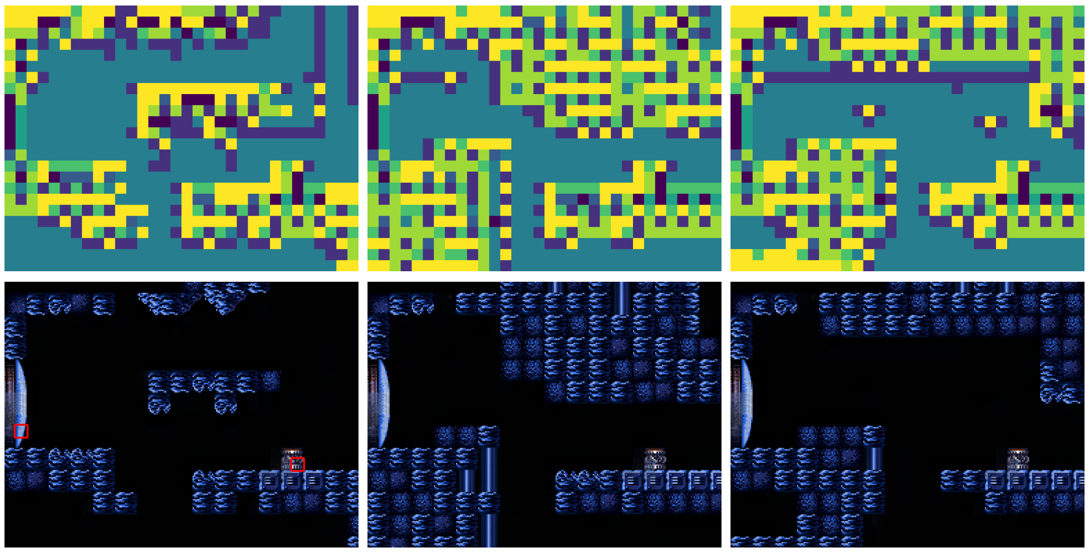


Figure 11.6: Constrained WFC + Neural Decoder outputs from *Super Metroid* data. One specific grid location has been constrained to have the appearance of a door tile while the other is constrained to be a power-up pedestal (outlined in red in left image).

11.4 Related Work

By positioning our approach as *neurosymbolic*, we contrast it with previous symbolic and neural approaches to map generation. This section briefly relates our work to these methods.

11.4.1 Symbolic PCG Methods

Maxim Gumin's WaveFunctionCollapse (WFC) [144] is an obvious precedent for this work. Karth and Smith [186] interpreted WFC as an instance of constraint solving methods for PCG, describing a rational reconstruction of WFC on top of the constraint-programming technology answer-set programming (ASP). Earlier, Smith and Mateas described a general approach to procedural content generation rooted in symbolic AI. In their paradigm, symbols and rules define a *design space model* that declaratively captures the space of all designs that might be appropriate to a scenario. Constraint-solving methods are then used to sample designs from this space that satisfy all modeled constraints. Where constraints are used in other PCG systems [108,160,327] they relate explicitly defined symbols or predicates operating on them. In this chapter, constraints over the placement of one tile next to another in the grid are handled by WFC.

Karth and Smith separately interpreted WFC as an instance of machine learning methods for PCG [187]. Under the assumption that the input was already composed of recombinable tiles, WFC trivially learns which tiles may be placed adjacent to one another. The idea of using adjacencies observed in the training data to influence adjacencies that will be observed in generator's output is also present in the multi-dimensional Markov chain (MdMC) work of Snodgrass and Ontoñón [331]. By contrast, most work in procedural content generation via machine learning (PCGML) has opted to use *neural* learning techniques.

11.4.2 Neural PCG Methods

A recent survey of deep learning for procedural content generation [209] reviews several neural approaches to example-driven PCG (Sec. 3.4.4). Among these, our current work makes useful contrast with past neural approaches to *Mario* level generation. Whether using a long short-term memory (LSTM) [343] or generative adversarial network (GAN) [367], these approaches directly trained a *neural generator* that used continuous representations during the sampling of discrete output tiles. While these approaches cleanly stayed within a single paradigm of AI, they limited the ability to directly and transparently apply constraints to the desired outputs.

Outside of procedural content generation for games, the larger computer graphics and computer vision communities are increasingly adopting *discrete* neural representation learning methods [361]. Even though image pixel data might be considered continuous, transforming it into inherently discrete representations allows methods originally developed for natural language processing (such as transformer networks [364]) to be cross-applied. In light of recent high-resolution image generation models that use learned discrete tile representations [104, 285], the use of WFC in our work shows constraint-solving methods as a symbolic alternative to the reasoning implicitly happening inside of transformer networks.

11.5 Future Work

Among a number of diverging next steps for this work, we wish to highlight two that tighten the integration between the neural and symbolic parts.

Improving manipulability and recombinability of symbols: In the history of AI [149], symbols were associated with representations that could be manipulated and recombined according to explicit rules, without consideration for the symbols' semantics. In the future, we might make the tile adjacency validity matrix (which functions as a nondeterminstic ruleset for WFC) be a trainable part of the neural model and use specific neural architectures or losses to encourage the matrix to have certain kinds of regularity or sparsity. We might also incorporate ideas from generative adversarial networks (GANs) to make sure that unseen-but-legal arrangements of latent codes produce rendered outputs that are similar to those seen in the training data. The goal is to make it so that any arrangement of symbols allowed under the extracted constraints leads to plausible renderings.

Richer symbol grounding: Beyond asking the autoencoder network to reconstruct an image of a game map design, we can ask the network to reconstruct additional data associated with the map (e.g. the precise tile data used by game engines, presence and properties of game objects on the map, historical player behavior data, etc.). This may allow reasoning about properties we want to constrain (such as reachability) using only the latent tile representation. We imagine these additional signals to be presented to the autoencoder model as additional feature channels beyond the red/blue/green pixel brightnesses currently modeled. By customizing the architecture of the decoder network (e.g. changing the receptive field of certain outputs), we can express that some inherent features of a tile should be decodable without any context whereas others might only emerge in consideration for broader regions. By grounding the latent symbols in a richer semantic frame, the resulting generator might also be able to output a high-level interpretation of a map design alongside the low-level data structures needed to represent that map in a target file format.

11.6 Conclusion

By training a VQ-VAE we can learn mappings for large source images, translating them into a latent representation. The latent representation is used as input into WFC's adjacency learning, defining the constraint rules that the solver satisfices. Once the solver has come up with an acceptable solution, we translate the new configuration of latent tiles back into the original pixel map via the VQ-VAE decoder, producing a context-sensitive rendering to the pixel image.

Constraint-based procedural content generation using symbolic methods has the advantage of being relatively easy to control, especially when the desired uses have hard constraints that can be specified as rules. In contrast, deep neural networks have demonstrated amazing generative abilities, but are difficult to control. This makes them much less useful in a production environment, particularly for videogames, in which many content types have constraints: an amazing level generator is not useful if the player cannot reach the end of the level. We demonstrate that a neurosymbolic approach, combining discrete representation learning methods with a constraint-based generator, allows the expression of large effective tilesets while offering enhanced controllability.

11.7 Acknowledgements

The original research, on which this chapter was based, involved myself, Batu Aytemiz, Ross Mawhorter, and Adam M. Smith. We jointly co-wrote the conference

paper version of this chapter [184].

Chapter 12

Vivisecting *Elite*

Having built up a general theory of procgen in three domains (poetics, generativist reading, and vivisecting the structure of a generative system in operation), we can now apply these lenses to an existing generative system.

The three research questions of this dissertation on aesthetics (RQ1), composition (RQ2), and application (RQ3) come together here to inform my analysis of *Elite* as a generative system, by means of a rational reconstruction.

One methodology for studying AI is rational reconstruction, "reproducing the essence of the program's significant behavior with another program constructed from descriptions of the purportedly important aspects of the original program" [266, p.235].

Rational reconstruction is particularly important for procgen as a methodology: while some of the significant generative systems are open-source, a greater portion of them are not: between closed-source commercial games and lost source code, there are thousands of generative systems we can't easily study.[1] Further, close reconstructions can sometimes tell us things about the system we are studying that the system itself cannot, particularly if supposedly inessential changes lead to the behavior differing in unexpected ways [266, p.245].

The generative system examined here was intended to be treated as a rational reconstruction at a distance, but during the course of development an unusually rich amount of information about the original system became available. As such, I have been able to test the reconstruction against the original behavior to a greater degree than I originally intended. Where the rational reconstruction is important is when we add additional capabilities to test generator configurations that would be impossible on the original hardware.[2]

12.1 Elite

Two of the most common games invoked in the "what is PCG" section of academic procgen papers are *Rogue* (1983) and *Elite* (1984), both frequently cited as early examples of procgen.[3] They are certainly both worth mentioning, but most mentions are perfunctory, and many confuse the motivations for using procgen.

The developer motivations are worth discussing, because the architecture for the generators was naturally shaped by those motivations. The lessons we can learn

[1]Many procgen algorithms have spread this way, by people operating off a description of a generative system and implementing it from scratch, sometimes discovering new approaches along the way. WaveFunctionCollapse is a prime example of this (Sec. 9.4.1), but is merely one recent example.

[2]My rational reconstruction can be found at `https://github.com/ikarth/elite`

[3]A very incomplete sampling: [8, 18, 19, 83, 133, 210, 291, 324, 344, 353, 383, 386]

from them are similarly colored by how the poetics of the generators we are studying or trying to construct differ from the poetics of the generators we are learning from.

While all generative systems have multiple motivations, there is often one that takes precedence. *Rogue* (as mentioned in Sec. 5) was motivated by a desire for *surprise*, including being able to surprise the developers themselves [375]. The most significant motivation for *Elite*, on the other hand, was *space*: not the outer space the players flew through, but the extremely tight memory limitations of the BBC Micro. They both, of course, received the benefits of the other motivation: *Rogue* takes up less space than it would otherwise, and *Elite* is capable of surprising the player. But each retains the effects of their particular emphasis.

The core gameplay loop in *Elite* is flying a spaceship. In three-dimensional wireframe graphics, no less, which by itself was an innovation worth attention. But it didn't stop at flying: the world the player is flying through is an open world, with the player free to choose what direction to go in, what planets to visit, and what activities to do. The larger gameplay loop was a commodity trading simulation, casting the player as a kind of space trucker or independent trading ship: buy low at one planet, hope to sell high at the next. But how the player goes about that isn't prescribed: you can be a peaceful trader, flying from port to port and dodging pirates along the way. Or you can start hunting those pirates for the bounty and whatever cargo they are carrying. Or, since no one asks where you got your cargo from, you can turn pirate yourself and hunt down other merchant ships. All of this can take place at any one of the hundreds of planets in the game, with the next planet just a hyperjump away.

Elite was developed by two Cambridge university students, David Brabin and Ian Bell, and published by Acornsoft in 1984 for the BBC Micro B. *Elite* didn't start as a procgen project. It started as a visualization of 3D spaceships that the pair of developers felt required gameplay that was more interesting than merely incrementing a score [337, 91-92]. Making a space sim with trading mechanics was an innovation, on a number of fronts. But trading needed places to travel between, and they ran into the strict limitations of the BBC Micro B:

> Their first idea had been to furnish the machine with the details of (say) 10 solar systems they'd lovingly handcrafted in advance: elegant stars, advantageously distributed, orbited by nice planets in salubrious locations, inhabited by contrasting aliens with varied governments and interesting commodities to trade. But it quickly became clear that the wodge of data involved was going to make an impossible demand on memory [336].

The game had to fit into 32,768 bytes of high-speed RAM, of which a third was already in use by the operating system. There wasn't room to store a universe, so the solution was to have the game generate everything just-in-time. And so it begins.

Elite has subsequently been a major influence on procgen in specific and games in general. One notable academic look at *Elite* is Alison Gazzard's platform studies article contextualizing *Elite*'s position in history [124]. In it, the generator is mostly mentioned in passing, rather than being a focus. However, it has a useful discussion of what the claims of *Elite* being first to do something mean in understanding the cultural context the game exists within, including its effects on later games and our present discourse.

12.1.1 Sources

A difficulty of analyzing generative systems in detail is that many of the most significant commercially-released generative systems are closed-source. While this doesn't preclude all analysis, it does make it significantly more difficult to analyze micro-details of the architecture: non-deterministic black boxes are infamously difficult to accurately model, after all.

Fortunately, the source code for *Elite* is available in full. Ian Bell released a number of files on his website,[4] including 6502 assembly source code for the original BBC version of *Elite* and a collection of different versions and ports of the game (the BBC cassette version, BBC 2nd Pro disk version, ports for other versions of the BBC Micro, Acorn Archimedes, Amiga, Amstrad CPC, Apple II, Atari ST, C64, Commodore Plus/4, IBM PC, MSX, NES, ZX Spectrum, etc.) and related documentation and ancillary files.

Notably, it also includes a translation of the universe generation and market trading from 6502 assembly into C. Ian Bell released this translation, entitled *Text Elite*, in 1999 [24]. *Text Elite* is useful as a reference implementation, because the original assembly code is in a particularly obtuse and cryptic format, necessitated by disk space and memory limitations. Other translations exist: Ian Bell explicitly mentions Christian Pinder's reverse-engineered Windows reimplementation as an influence on his own public release.

Christian Pinder's *Elite: The New Kind*[5] is only one of several modded or

[4]`http://www.iancgbell.clara.net/elite/`
[5]=http://www.new-kind.com/ [271]

reverse-engineered versions of the original *Elite*. Angus Duggan's *Elite A*[6] is a modded version of the original BBC *Elite*, adding features like flying new ships, special cargo missions, an in-game encyclopedia, among other enhancements. *Oolite*[7] is a separate open-source implementation inspired by *Elite*, for Windows, Mac OsX, and Linux. It uses an algorithm very close to the *Elite* generator to do its world generation, though with some differences.

Indeed, all of the various ports and versions have slightly different behaviors—some minor, others significant. While most ports made an effort to keep the core of the algorithm the same, discrepancies in the individual platforms led to slightly different effects in edge cases.

Finally, and most usefully for our purpose, Mark Moxon has published a complete annotated account of the Elite source code, annotating each line of the source code and comparing changes across different versions of the game.[8] Moxon credits earlier disassembly and annotation work by Paul Brink [9] and Kieran Connell,[10] and assistance from Angus Duggan, Christian Pinder, and Chris Jordan. Moxon's fully documented version of *Elite* is a remarkable achievement and makes *Elite* one of the most thoroughly documented early videogame codebases. While the rational reconstruction described in this chapter was begun without access to most of the above-listed resources, constructing and error-checking the final version would have been much more difficult without

[6]url=https://www.bbcelite.com/elite-a/ [92, 250]

[7]url=http://www.oolite.org/ [377]

[8]`http://bbelite.com` [245]

[9]url=http://www.elitehomepage.org/archive/a/d4090010.txt [42]

[10]`https://github.com/kieranhj/elite-beebasm` [68]

them.

The sequels to *Elite* (*Frontier: Elite II*, *Frontier: First Encounter*, and *Elite: Dangerous*) constitute a second lineage, with a different approach to the architecture of the generator. As such, I won't be discussing them here in detail, though their attempts to simulate the entire Milky Way galaxy within their hardware limits are noteworthy and rate their own future discussion.

12.1.2 The Seed

Elite's generators are peculiar when compared to modern approaches. This is, primarily, because they were engineered to be extremely compact in keeping with the limitations of the BBC Micro. Secondly, the approach to the random number generation is unusual from a modern perspective and consequently has a unusually-shaped latent space.

The core of the generator, and also its primary source node, is the random number seed. And, of course, the process that transforms that seed into the next in the sequence. Having a random seed is ordinary, but the way it is used is not. In a modern context, the typical way to use a random number generator is to use it indirectly, transforming it into some number in a given statistical distribution. Instead, the bits in the seed's six bytes are used directly.[11] The planet's government type, for example, is defined by a three-bit value, extracted directly from three bits in the low byte of the second word of the three-word planet seed.[12]

[11] `https://www.bbcelite.com/deep_dives/galaxy_and_system_seeds.html` [247]
[12] `https://www.bbcelite.com/deep_dives/generating_system_data.html` [248]

This kind of direct use of bits characterizes the compact nature of the code and is also responsible for the particular shape of the generator's latent space. This is compounded by the weaknesses of the random generator itself: in technical terms it is a weak PRNG that is relatively predictable. In particular, some of the individual bits are prone to bad behavior that warps the latent space. One side effect is that the fourth galaxy only features a subset of the possible galaxies: in that galaxy the last bit is always 0, so only the (zero-indexed) 0th, 2nd, 4th, and 6th governments are generated. (No dictatorships allowed!)

Here we see why the motivation is important: the generator is simple, so it doesn't take up much space on disk, and it is fast, leaving more of the precious time for the 3D graphics calculations. A more robust PRNG would lack that critical property. As a side effect, the biased nature of the PRNG resulted in galaxies that occupy a particular subset of the possible latent space; the expressive range has been confined, giving the game's galaxies their personality.

From a design standpoint we'd probably like to have more control over the shape of those personalities. However, on a resource-constrained computer, having that embedded into the emergent latent space is an effective use of resources, whether intended or not.

For most deterministic generators, the seed is the axis the rest of the generator revolves around. For *Elite*, in a sense, the seed *is* the generator: the individual bits are directly translated into the generated planet attributes. Advancing the seed to the next

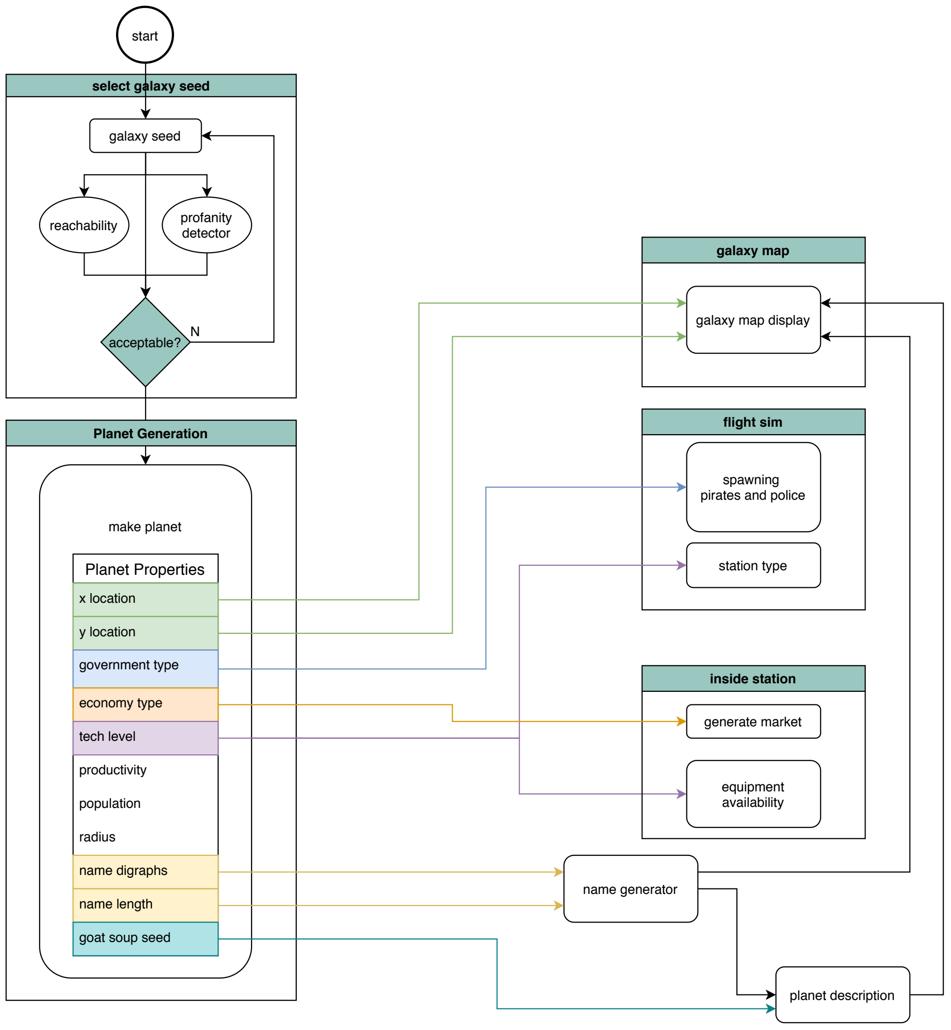


Figure 12.1: Diagram of the *Elite* generative system. The major components are the selection of the galaxy seed, the system generation (see Fig. 12.3), and the planet description generation. The galaxy seed selection was done offline, using a tool that allowed the programmers to filter out the unacceptable galaxy seeds. The other components were regenerated just-in-time rather than being stored in memory (in the original BBC Micro version of the game). Star systems in the original *Elite* always have exactly one star and one planet. Other planet values are used when flying through the system to control pirate spawning and using a visually different space station model in high tech systems

state gives you the next planet's seed.[13] In hexadecimal, the three words of the official starting seed are 5A4A 0248 B753. There are many possible seeds, but the developers chose this particular seed after an extensive search.

The seed was not chosen arbitrarily: the developers built a tool to enable them to search for a starting seed that would fulfil the two major criteria they had [37]. First, all of the systems in each galaxy needed to be spread relatively evenly without clumping so that each system is reachable from every other system. Second, it shouldn't include rude or profane words that would get them into trouble: In one of early seeds they tried, the planet name generator rudely and unhelpfully named one of the planets "Arse," leading to what is probably one of the earlier examples of discarding a game's generative result because it included unprintable artifacts [337, p.104]. Using this tool, they examined a "beauty parade" of 100s of candidates [37].

This search program is an early example of a generate-and-test loop, forming a topological cycle when diagramming the generator (Fig. 12.1). It is also an offline process, run and completed before the game itself was finished. While we usually emphasize the just-in-time nature of Elite's online generation, it is important to understand that the latent space of the online generator was extensively shaped and constrained by the prior offline generative process.

The inclusion of the search program also emphasizes that the generator can't be understood as a monolithic object that can be sorted into our existing taxonomies of generativity: taken as a whole, the generator is *both* online and offline. An analysis of the

[13]Technically, the next planet seed is arrived at after 'twisting' it four times.

online generator is incomplete without considering the shadow of the offline generator. This is a common pattern in development, both at the micro level as developers tweak the generator by observing the artifacts it produces while they are constructing it, but also at the macro level. For example, Spore shipped with a list of vetted planet seeds that had been checked for problems like broken slopes due to too-rapid changes in the heightmap [62].

12.1.3 The Random Generator

The algorithm for transforming the seed is fairly straightforward. It involves executing a "twisting" operation inspired by a generalization of the Fibonacci sequence [37] with three little-endian 16-bit numbers instead of two natural numbers: $A_1 = B_0; B_1 = C_0; C_1 = (A_0 + B_0 + C_0) \mod 65536$ [249]. The 16-bit limitations mean that it will wrap, of course, but this is enough uniqueness for the purposes of the game. This random number generator is used for every random element in the game, from the position of stars to drawing explosions [244].

This three-term pseudo-random number generator is fairly similar to known PRNGs, such as Lagged Fibonacci Generators [221]. For example, Green, Smith, and Klem described a similar two-tap additive RNG in 1959 [134]: $X_j = (X_{j-i} + X_{j-n}) \mod 2^r$. Though most discussions seem to concentrate on two-tap generators (with the occasional generalized exception, e.g. [39]), whereas the *Elite* generator is a three-tap generalized Fibonacci sequence.

12.1.4 Planets

To get the seed for the next planet, twist the seed four times. The zero-th planet uses the galaxy seed without a twist, the tenth planet twists forty times. To get to the next galaxy, roll the bits to the left, yielding eight different starting seeds that loop around back to the first galaxy.

When the player jumps to the next galaxy, the original game places them at the system closest to $96, 96$ [24, line 643]. There is a bug here in the BBC Micro cassette version.[14] Later versions explicitly set the player to the system location directly, but the cassette version just places the player exactly at $96, 96$, potentially stranding them if there isn't a system within fuel range. In either case, if there are any systems in the galaxy that can't be reached by a chain of 7.0 light year jumps, they are unreachable by conventional means (cf. Sec. 12.3).

Each of the planet's attributes is generated on the fly, populated directly from different bits of the seed, and sometimes other attributes [248]. Contemporary practice would be to use the seed to do a series of die rolls (or, technically, sample from the random distribution), which would make the whole process easier and more controllable. However, using the seed directly means that the calculation for a particular planet only needs to be run *once*, rather than per-attribute.

Speed is particularly important because most of the planet data isn't stored for very long. The game is divided into the flight module and the docked module. The

[14]Helpfully documented by Mark Moxon: `https://www.bbcelite.com/cassette/main/subroutine/ghy.html`

only data that the flight model keeps in memory is the planet's economy, tech level, government, and availability of items on the market. Everything else is calculated as needed. The planet description is an extreme example: as the sequence characters is in the middle of being printed on screen, the parser expands the grammar rule tokens as it encounters them [246]. The expanded version of the entire description basically never exists in memory.

```
(defn planet-government
  "Planet government is a number from 0 to 7, extracted directly from the
     ↪ bits in the seed.
  The first operation in the original code, and the most basic."
  [planet-seed]
  (extract-bits planet-seed :s1_lo 2 3))    ; start at index 2, get 3 bits

(defn galactic-x [seed]
  (extract-bits seed :s1_hi 0 8))

(defn galactic-y [seed]
  (extract-bits seed :s0_hi 0 8))
```

Listing 12.1: Some of the planet generation functions from my rational reconstruction of *Elite*'s generator. While the generator's artifacts are identical to the original, the reconstruction includes some additional behavior, giving it greater flexibility and more error checking (which was unnecessary in the linear-sequence original generator). There are also a lot of direct reimplementation of bit-manipulation operations, such as (extract-bits) here, to ensure that the outcomes exactly emulated the original behavior and limitations of the BBC Micro.

12.1.5 Grammars

Elite is a relatively early example of a game using a procgen grammar. The grammar is, in many ways, a natural extension of the game's basic text printing routines. Rather than storing text as strings of ASCII characters, the text data is represented by sequences of tokens, which represent anything from individual printable characters, to

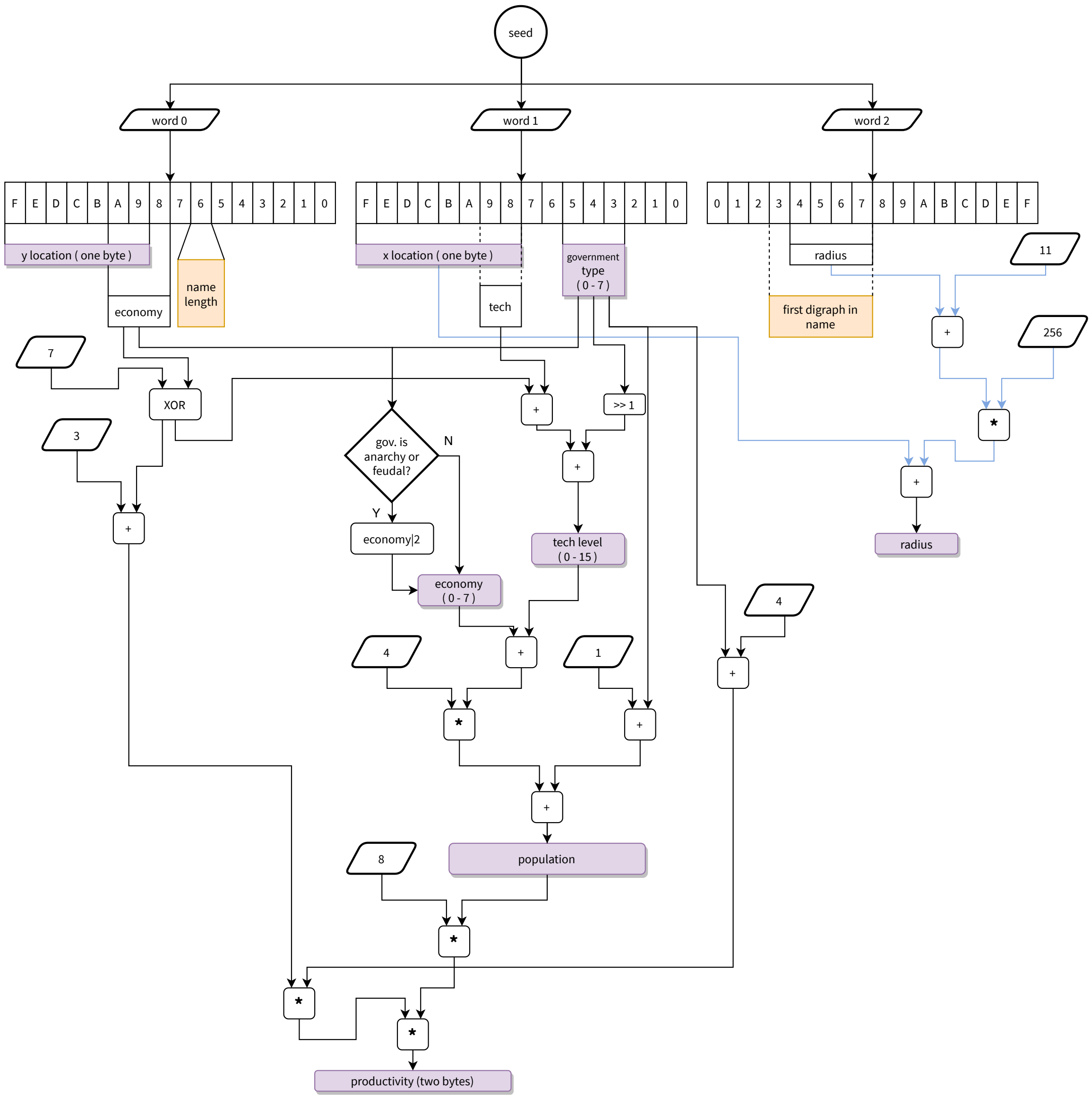


Figure 12.2: Visual diagram of *Elite*'s system generator. In contrast to the way more recent generators tend to use random number generators, *Elite* directly uses values from the current seed. Shaded nodes indicate the artifacts that are output from the system generator, illustrating how artifacts can be generated along the way and incorporated into the context of other generated artifacts. The diagram is intentionally more low-level than would usually be practical for casual analysis, in order to more clearly demonstrate the relationships between the seeds and the extracted bits. For a more practical vivisection, consult Fig. 12.3.

recursive tokens that can contain lengthy sequences which can themselves contain more recursive tokens.[15] The tokens are each single byte.[16] This both saves space and makes the grammar possible. The two letter tokens, for example, allow you to store "`LASER`" in the three tokens "`149 S 144`" but are also used by the planet name generator to glue together any of the pairs and turn "`149 150`" into "`LAVE`". Since the planet names are generated with two-letter tokens—with the exception of token 143, which is just the single letter 'A'—so, unless the name has an A in it, names can only have an even number of letters [252].

To reiterate a significant point about the grammar: the planet description *never* completely exists in memory because each token is expanded *while it is in the middle of being printed to the screen.*

Some parts of the grammar allow for randomized expansion. Each randomization token in the grammar eventually leads to the start of a five-token block in the grammar. When the generator encounters a randomization token, it picks one of the five using the next random number in the PRNG. As usual with a chain of linked PRNG invocations, any alterations to the earlier elements change the later ones.

The tightly-compacted generator has biases, some intentional and some not. For an intentional example, the way the randomizing elements in the species description look up the corresponding adjectives with a sliding window on the token table means

[15]As Mark Moxon usefully unpacks [252]: `https://www.bbcelite.com/deep_dives/printing_text_tokens.html`

[16]Advanced versions of the game also added an extended token table, accessed via a different subroutine, this is where the lengthy system description grammar resides [243]

that the set of adjectives is dependent on the species: no slimy cats.[17]

12.1.6 Expressive Range

There are 2^{48} seeds possible with three 16-bit integers. The expressive range is vastly smaller: due to the way the generator uses a subset of the seed, many of the possible configurations are unused, while others duplicate each other. These limitations show up in the released game: in the fourth galaxy, the last bit of the three-bit government parameter is always 0, so only half of the government types are generated.

The developers initially planned to use the entire 2^{48} range. Their publisher, Acornsoft, advised them to limit it to a more reasonable number: "the bigger the world, the more you understood that this was something being spread successively thinner" [337, p.104].

In the end, they settled on eight galaxies, for a total of 2048 stars.

There are several factors that conspire to make the generated galaxies feel more solid: the fixed seeds, the ritual of travel, the changing local conditions, and the emergent properties of the particular configuration of each galaxy.

The fixed seed means that you are going to revisit the same stars often. You can't quickly re-roll for a new galaxy; you have to live with the one you've got. This is not usually what we associate with procgen, but it is, I think, a significant aesthetic undercurrent that informs the feel of many generators: they all have *some* fixed struc-

[17]It's quite compact and clever, as Mark Moxon's commentary shows: `https://www.bbcelite.com/deep_dives/generating_system_data.html`

ture, and embracing that is more effective than futile rejection.[18] There are generative aesthetics that can only be accessed by being locked out from the gambling addiction of one more pull of the generator's arm.

Likewise, the ritual before seeing a new artifact makes each artifact have more meaning. The difficulty of travel makes it take longer between seeing different stars: the ritual of pulling away from the station, hyperjumping, flying or fighting to the next station, and docking creates a real cost. A cost that is measured in both the player's time and their in-game fuel costs. Elite was a bit unforgiving even when it was released, and just the 256 stars of the first galaxy will keep you occupied for a long time.

Each time the player hyperjumps into a new star system, a unique-to-this-visit seed is generated. The long-term practical effects of this seed are mostly confined to the current market price and availability of the commodities for sale. Therefore, the significant changes between different trips to the same planet are mostly confined to these market conditions (and perhaps the number of pirates you happened to encounter this time). While small, this change over time prevents the trading profits from being a sure thing. This emphasizes the comparison of different visits, and therefore this juxtaposition across time informs the aesthetic effect. Changing an existing artifact is sometimes more effective than creating a completely new one: the player is already familiar with the original, and can therefore better recognize and appreciate the differences (Sec. 5.8).

The configuration is an accident of the generator, albeit a planned-for accident: during the search for a good starting seed, the shape of the galaxies was considered.

[18] I reiterate: all information in a generator needs to come from *somewhere* (Sec. 8.1.2).

The distribution of stars, crossed with the limited range of the jump drive, creates an emergent network of trade routes, with clusters and bottlenecks that shape the players' options.

A handful of stars are impossible to reach in the original game by the usual means: in the original version of the game, a galactic hyperjump always placed you in the system nearest to coordinates 96, 96, which meant that any systems that weren't part of the same group of stars were unreachable (Sec. 12.1.4). In practice, this is only a handful of stars, and later versions of the game exhibit slightly different behavior.

Because the player's trading profits depends on the differences between nearby systems, the local composition of a cluster of stars matters. The trading player is looking for a profitable route of nearby stars with different economies and relatively safe travel conditions. Computers are cheap at industrial planets and expensive at agricultural ones, for example. Though you also have to take into account the fluctuating prices, the price of fuel, the risk of encountering pirates, the legality of the cargo, and that there isn't a guarantee of that commodity being in stock in the first place. It's a relatively simple economic model, but one that prevents the player from getting too complacent too quickly.

12.2 Vivisecting the Planet Generator

Diagramming the vivisection of the planet generator is relatively straightforward (Fig. 12.3): The planet seed acts as the *source*, feeding into several different

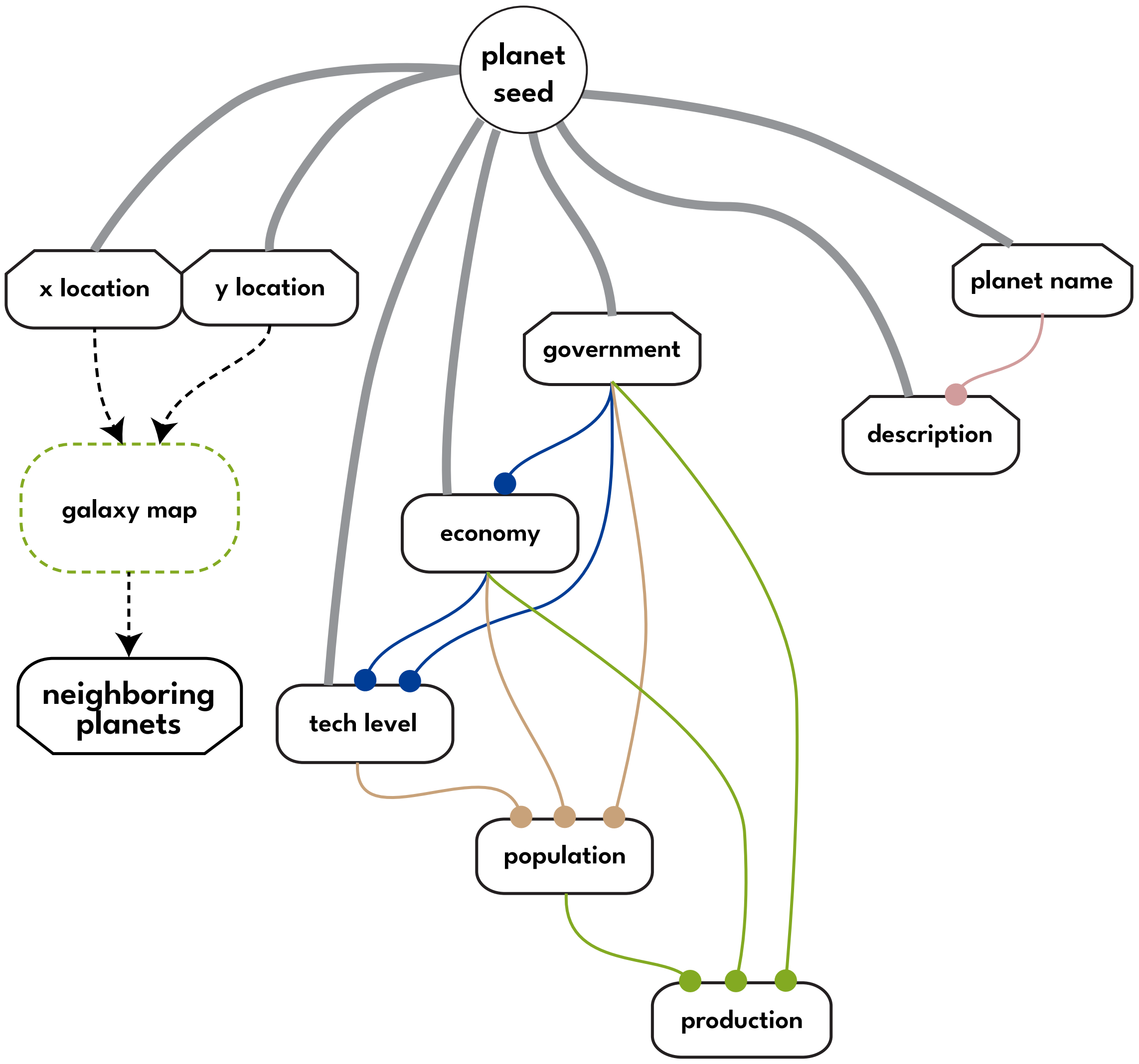


Figure 12.3: The *Elite* planet generative system, modularized into its main generative operations (cf. Chap. 8). The *source* planet seed feeds into several different *generator transformation processes*, which in turn feed into *translation transformation processes*. The galaxy map is implicit, only existing as the aggregate result of the star's coordinates. The original game only makes use of this indirectly; if we want to make use of it for our reconstructed generator we need to add a *reading transformation process* to convert from the collection of coordinates to a list of nearby stars.

generator transformation processes (primarily the government type, the planet name, and the x/y coordinates), some of which feed into chains of *translation transformation processes*.

The original generator doesn't include any *reading transformation processes*: part of the reason it is so fast is because all of its readings are implicit and embedded in the relationships between the various attributes of the planet. Some variants, such as *Oolite*, do track some additional information, such as the number of reachable nearby planets, making this information available to mods. Therefore, in my rational reconstruction I've included a reading transformation process that looks at the implied galaxy map and lists the nearby stars.

Sinks are implied. Technically the output of any of these operations can be used as an artifact: most of them do feed into other parts of game, and all of them are displayed as part of the planet description screen. A generator for a game that is less resource-constrained might include some hidden operations whose output is only fed into other operations and never used directly. But *Elite* doesn't have the resources to spare: every byte saved is another byte that can be used for major game mechanics.[19]

There are several useful features we can see at a glance: the seed is used directly by many (but not all) parts of the generator; there are a couple of chains of operations that feed into each other; and the planet description is entirely decoupled from everything except the planet's name. This modeling approach also has its limits: the description might still be interrelated with the other variables if the way it uses the

[19]Fuel scoops, for example, used four bytes of memory but led to an entire chain of game mechanics: the player can use them to collect cargo, which is what made piracy possible. [337, p.107]

seed means that some attributes might be impossible or unlikely to be paired with some descriptions.

With *Elite*'s descriptions such a relationship would be accidental in *most* cases, though flaws in the random number generator could still cause an unexpected correlation. But beyond that, the limitations of using the bits directly from the seed mean that if different operations draw from the same bits they will have the same cause, and therefore will *definitely* be correlated. It happens that the planet name generation and the planet radius use overlapping bits in the seed—meaning that there is a direct, if obscure, correlation between the name of a planet and its radius.

The original *Elite* mostly only uses the planet radius for flavor text, so the consequences for this are slight. But the peculiarity of this side effect gives me the opportunity to re-emphasize that looking at just the obvious parts of the generative system doesn't capture the whole picture. Information is embedded into the processes themselves and the topology that they're assembled into. These invisible relationships encode a lot of what goes into making the actual artifacts.

```
(defn export-galaxy [galaxy-seed galaxy-index]
  (let [jump-range 70
        this-galaxy-planet-list (planet-seed-list galaxy-seed)
        indexed-planets
        (map-indexed (fn [i p] [i p]) this-galaxy-planet-list)
        planet-coord-list (map-indexed (fn [index p] [index [(galactic-x p)
            ↪ (galactic-y p)]]) (planet-seed-list galaxy-seed))
        planets-in-galaxy
        (for [[index p] indexed-planets
              :let [name (last (take 7 (iterate generate-name (
                  ↪ generate-name-start p))))
                    gov  (planet-government p)
                    econ (planet-economy p gov)
                    tech (planet-tech-level p econ gov)
                    popl  (planet-population-size tech econ gov)
                    prod (planet-productivity econ gov popl)
                    galactic-x (galactic-x-uniform-random p galaxy-index
                        ↪ index)
                    galactic-y (galactic-y-uniform-random p galaxy-index
                        ↪ index)
                    reachable-systems (list-reachable-systems [galactic-x
                        ↪ galactic-y] planet-coord-list jump-range)
                    hub-count  (count reachable-systems)
                    description (planet-goat-soup p name)
                    trade-routes
                    (map
                     (fn [other-data]
                       (let [other-planet (nth this-galaxy-planet-list (nth
                           ↪ other-data 0))
                             other-gov (planet-government other-planet)
                             other-econ (planet-economy other-planet
                                 ↪ other-gov)]
                         (calculate-trade-route-value (second econ) (second
                             ↪ other-econ) (nth other-data 0))))
                     reachable-systems)]]
          {:index       index
           :name        name
           :species     (planet-species p)
           :government (government-name gov)
           :economy     (economy-name econ)
           :tech-level tech
           :population popl
           :productivity prod
           :location    [galactic-x galactic-y]
           :neighbors  reachable-systems
           :neighbor-count hub-count
           :description description
           :trade-routes trade-routes})]
    planets-in-galaxy))
```

Listing 12.2: A straight-ahead configuration of the *Elite* planet generator, showing how the different parts of the system feed into each other. Includes the additional trade-route calculations, which are not in the original.

12.3 The Poetics of the Trading Lanes

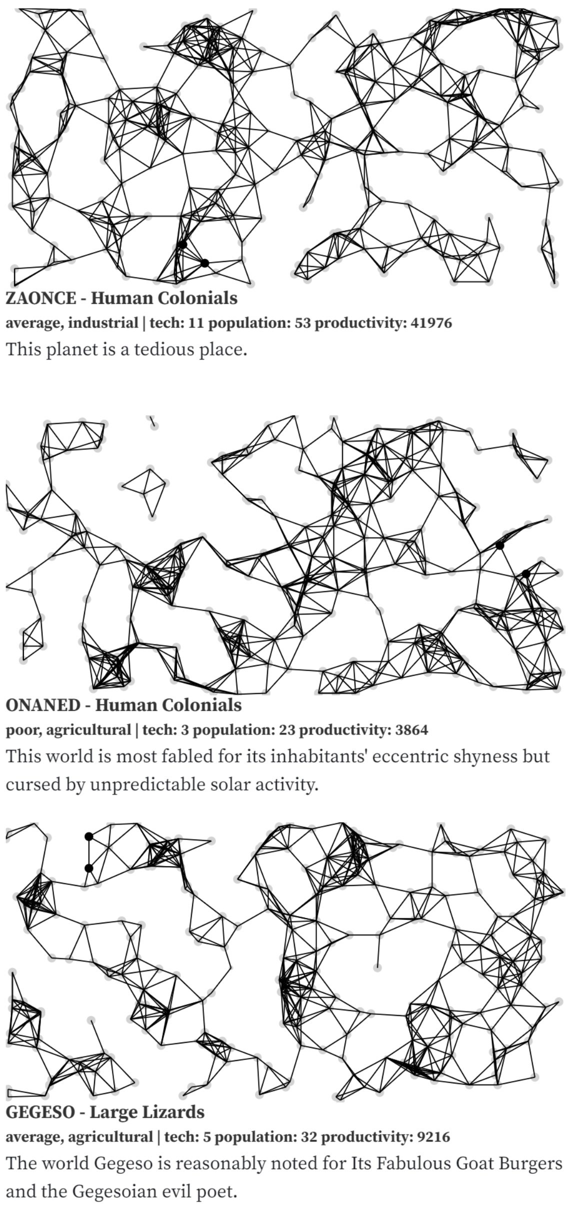
The x and y coordinates use entire bytes from the seed. Therefore, the shape of the galaxies in *Elite* is an entirely unfiltered consequence of the way the random number generator works.

An indirect effect of this is the direct trade routes for each planet. The number of neighboring planets is an implicit attribute in the original *Elite*: the local neighborhood is an emergent effect of the way stars are scattered. But the trade routes that emerge from the stars which the player can reach are what shape the medium-to-long-term gameplay. Some of the games inspired by *Elite*, such as *Oolite*, do keep track of this and make it available for modding the game.

In *Elite*, Your ship has a maximum hyperjump distance of 7 lightyears, though the way the hyperjump distance is calculated depends on the version of the game (Sec. 12.1.4). Different versions resolve the bug in different ways (Fig. 12.5).

Each of the commodities that can be traded has a base price, which is multiplied by the planet's economic type, with agricultural and industrial economies having different patterns of demand. The amount currently in stock to buy is calculated similarly. In both cases, the base value is adjusted by adding a random seed, a unique value that is reset during the hyperjump calculations each time the player enters a new system. Interestingly, each commodity has an 8-bit mask that is applied to the visit seed with a bit-and operation. This gives each commodity a different level of volatility, so the amount it can swing also needs to be taken into account (Lst. 12.3).

The size of the player's 20-ton cargo hold (35 tons, if you purchase the upgrade); the price of the commodity; the amount of cargo space the commodity takes up (e.g. gold is measured in kg rather than tons); and the amount that is available to buy all interact together. This makes the trading profits dependent on the player's current resources, with the optimal choice changing as the player's capabilities change.



ZAONCE - Human Colonials

average, industrial | tech: 11 population: 53 productivity: 41976

This planet is a tedious place.

ONANED - Human Colonials

poor, agricultural | tech: 3 population: 23 productivity: 3864

This world is most fabled for its inhabitants' eccentric shyness but cursed by unpredictable solar activity.

GEGESO - Large Lizards

average, agricultural | tech: 5 population: 32 productivity: 9216

The world Gegeso is reasonably noted for Its Fabulous Goat Burgers and the Gegesoian evil poet.

Figure 12.4: Understanding the emergent behavior of *Elite* via a visualization of the trading routes in galaxies 0, 2, and 6. Dots and lines represent stars within 7 in-game lightyears of one another. While most of the stars are connected in a sprawling network, a handful are unreachable. A description of one planet from that galaxy is shown beneath each map. Data visualization courtesy of Jasmine Otto, used with permission.

All of this taken together means that while planets have a known characteristic (it's not usually profitable to import food to an agricultural world) they also have enough uncertainty that the player usually can't find a local maxima and simply cycle back and forth between two systems. The combination of the commodity mask, random seed, and planet attributes make the generative possibility space of the market prices just unpredictable enough. Thus, the player is encouraged to go out and keep exploring. The random noise has an interesting shape, one which shapes the player's gameplay choices.

The other major influence on the trading game is the frequency of pirate attacks, which are influenced by the government type. This acts as a separate gradient, related to, but separate from, the economic effects. Thus additionally partitioning the galaxy by the player's skill at fighting, or at least escaping.

The combination of the two influences is why the planet generator is so effective, for all its relative simplicity: because each planet has several generated attributes that tie into the rest of the game, those small differences matter. Elite has three scales it can display: the current system, the local neighborhood, and the entire galaxy. And at each scale, the way the generator matters is slightly different. Stay in one or two systems and the individual differences are what is important: Zaonce is a Corporate State at Tech Level 11—few pirates, good place to buy computers. Zoom out a bit and start comparing planets in the local neighborhood to each other, and the small differences become important. You see an orchard of planets, where we start to notice the patterns. Zoom out to the whole galaxy and survey it: individual stars matter less than how they are arranged as a group. The clusters and choke-points matter as you try to work out a path through the forest of stars to the distant high-tech planet which sells the galactic hyperdrive that will take you to the next galaxy.

The reason why the generator in *Elite* had such impact is that it parceled out its implications across these different scales. As the player trades, the commodities they care about change and the market values have different meanings. As they get more skilled at fighting (or buy better lasers) the safety of the planets becomes less of a necessity and more of an educated gamble. As they explore more of the galaxy, the overall structure comes into view as an invisible network of paths through the sky.

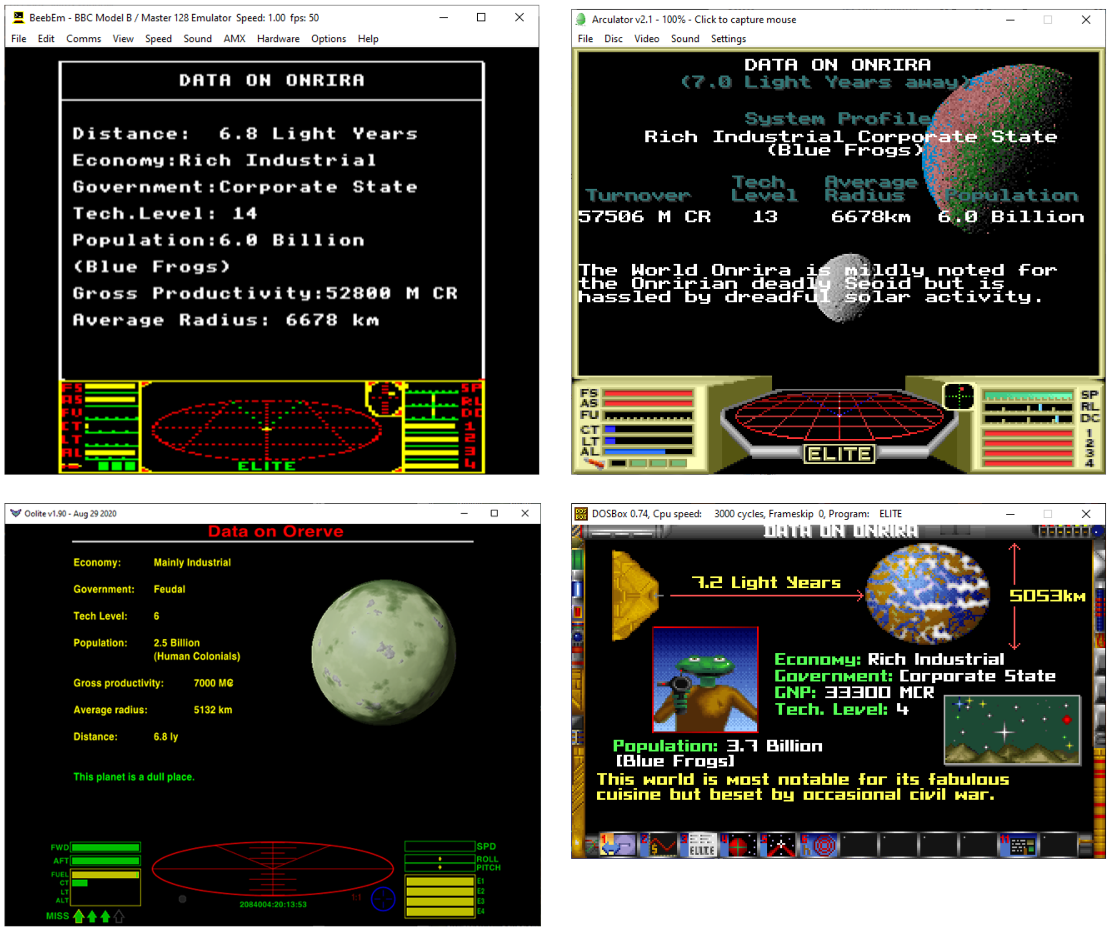
The generative system itself doesn't change, but because of the gameplay the player's relationship to the generated artifacts changes. It is this texture that led to the game being remembered.

```
(def commodities
  ;; Base price, economic factor, unit, base quantity, mask, name, legality
  [[19,  -2, 't',   6, 2r00000001  :food          :legal]
   [20,  -1, 't',  10, 2r00000011  :textiles      :legal]
   [65,  -3, 't',   2, 2r00000111  :radioactives  :legal]
   [40,  -5, 't', 226, 2r00011111  :slaves        :illegal] ;; illegal
   [83,  -5, 't', 251, 2r00001111  :liquor        :legal]
   [196,  8, 't',  54, 2r00000011  :luxuries      :legal]
   [235, 29, 't',   8, 2r01111000  :narcotics     :illegal] ;; illegal
   [154, 14, 't',  56, 2r00000011  :computers     :legal]
   [117,  6, 't',  40, 2r00000111  :machinery     :legal]
   [78,   1, 't',  17, 2r00011111  :alloys        :legal]
   [124, 13, 't',  29, 2r00000111  :firearms      :illegal]
   [176, -9, 't', 220, 2r00111111  :furs          :legal]
   [32,  -1, 't',  53, 2r00000011  :minerals      :legal]
   [97,  -1, 'k',  66, 2r00000111  :gold          :legal]
   [171, -2, 'k',  55, 2r00011111  :platinum      :legal]
   [45,  -1, 'g', 250, 2r00001111  :gem-stones    :legal]
   [53,  15, 't', 192, 2r00000111  :alien-items   :unavailable]])

(defn calculate-market-price [commodity-number economy visit-seed]
  (let [commodity (apply hash-map (interleave [:base-price :economic-factor
     ↪ :unit :quantity :mask :key] (nth commodities commodity-number)))
        base-price (+ (:base-price commodity) (bit-and visit-seed (:mask
            ↪ commodity)))
        econ-factor (* economy (abs (:economic-factor commodity)))
        final-price (* (+ base-price econ-factor) 4)]
    final-price))

(defn calculate-market-available [commodity-number economy visit-seed]
  (let [commodity (apply hash-map (interleave [:base-price :economic-factor
     ↪ :unit :quantity :mask :key] (nth commodities commodity-number)))
        base-amount (+ (:quantity commodity) (bit-and visit-seed (:mask
            ↪ commodity)))
        econ-factor (* economy (abs (:economic-factor commodity)))
        final-amount (rem (- base-amount econ-factor) 64)]
    (max 0 final-amount)))
```

Listing 12.3: The market economy calculations. The data from the table of commodities is used to calculate the current market price and amount in stock, based on the planet's attributes and a random seed that is unique to each visit to the system (reset when entering a new system via hyperjump).


BeebEm - BBC Model B / Master 128 Emulator Speed: 1.00 fps: 50
File Edit Comms View Speed Sound AMX Hardware Options Help
Arculator v2.1 - 100% - Click to capture mouse
File Disc Video Sound Settings
Oolite v1.90 - Aug 29 2020
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: ELITE
ELITE

```
DATA ON ONRIRA

Distance:  6.8 Light Years
Economy:Rich Industrial
Government:Corporate State
Tech.Level: 14
Population:6.0 Billion
(Blue Frogs)
Gross Productivity:52800 M CR
Average Radius: 6678 km
```

```
DATA ON ONRIRA
(7.0 Light Years away)

System Profile
Rich Industrial Corporate State
(Blue Frogs)

Turnover     Tech    Average   Population
             Level   Radius
57506 M CR    13     6678km    6.0 Billion

The World Onrira is mildly noted for
the Onririan deadly Seoid but is
hassled by dreadful solar activity.
```

```
Data on Orerve

Economy:           Mainly Industrial
Government:        Feudal
Tech Level:        6
Population:        2.5 Billion
                   (Human Colonials)
Gross productivity:     7000 M₵
Average radius:         5132 km
Distance:          6.8 ly

This planet is a dull place.
```

```
DATA ON ONRIRA

7.2 Light Years        5053km

Economy: Rich Industrial
Government: Corporate State
GNP: 33300 MCR
Tech. Level: 4

Population: 3.7 Billion
 [Blue Frogs]
This world is most notable for its fabulous
cuisine but beset by occasional civil war.
```

Figure 12.5: Examples of distance discrepancies on various versions of *Elite*, as demonstrated by the distance between Onrira and Orerve. Shown in clockwise order from top left: BBC Elite (6.8 LY), Acorn Archimedes Elite (7.0 LY), PC Elite Plus (7.2 LY), *Oolite* (6.8 LY). Not shown: TXTELITE (7.0 LY), PC Elite (7.2 LY)

12.4 Ecological Generator

Thus far we've mostly been considering the generative operations as being fixed in place in a rigid pipeline, with a passing mention of the possibility of generating a generative system (Chap. 8, Sec. 8.5).

A problem with pipelines is that they involve complicated and possibly mutually-exclusive dependencies. This is illustrated by what has been termed the *problem of orchestration* in game generation [207]: different parts of a game depend on each other, and there's no right order to generate them in. Generating a game based on a title requires a very different pipeline from generating a title based on a game.

Generative systems that change over time are also difficult to represent as pipelines. *Livecoding* is the practice of creating generative music by programming it live, as part of a performance. While a snapshot of the performance at a given point might be a pipeline, when a livecoder modifies a running generative system the performance as a whole is difficult to represent as a pipeline [319].

One alternative to the pipeline approach is what I've termed an *ecological generator*, so named both because of being composed of a rhizomatic assemblage of design-move organisms and because it was partially inspired by Anna Tsing's *The Mushroom at the End of the World* [359], in which the examination of interspecies entanglements in polyphonic assemblages is extended to the many different relationships that wave in and out of involvement with each other.

There are a number of influences that led to the ecological generator as a concept, but there are three that are most significant. The first, as noted, is *Mushroom at the End of the World*. Second, the discussions I had with Michael Mateas about his Content Selection Architecture. Lastly, I am indebted to Max Kreminski for the discussion, advice, and programmed example that contributed to the core orchestration design.

12.4.1 Design Moves

In the ecological generator, generative operations do not have fixed sources for their input parameters. Instead, they are encapsulated in *design moves*[20] which we treat as organisms in an ecosystem: a design move has a set of data types it can consume (the parameters for its generative operation) and a set of data it produces (the output of the generative operation). Rather than being permanently attached to a specific other generative operation node in the pipeline graph, it looks on the shared blackboard for data that matches its input type.

These design move organisms can be implemented as agents, but that isn't an inherent part of the definition: some of my implementations simply make them available as actions in a mixed-initiative interface. Design moves have the advantage of corresponding to the decisions that a human designer would make if they were constructing a generator.

Compared to simply designing a generative operation, designing a design move requires a tighter specification of what type of data it can accept. This is to be expected:

[20]Thanks goes to Max Kreminski for the terminology

the topology of a generative pipeline implicitly encodes a lot of information, and to write a design move we need to explicitly describe that information. All is not lost: in return, the way the design move senses the data it can accept can be arbitrarily sophisticated. My implementations use Datalog as an expressive query language, but you could equally use anything from simple decision trees to deep learning.

Determinism requires some extra thought: because design moves have no inherent order, if you want to get a consistent output you have to be careful about which design move gets run next. The pipeline encodes information about timing, too. In practice is isn't quite as much of an issue as it might appear: because most generators have to be careful about determinism anyway, if the design makes careful use of seeds and hashes the result will already be independent of execution order. Writing generative operations in an immutable, functional style with no generator-affecting side effects is, in my opinion, a good idea anyway. It remains something to keep an eye on, as naive implementations of generative systems can accidentally run afoul of this.

Alternately, you can deliberately construct design moves that take advantage of the execution order to make complicated live-coding-like changes to the generative system. In which case you will still want to have some kind of order priority to be able to replicate a set of design moves—possibly via a hash of the current blackboard state.

We can also capture the order that design moves are executed in and freeze their generative operations into a pipeline, chaining them together. An ecological generator is a flexible way to generate pipeline-based generative systems, making it useful as part of a generative-system-building tool. The way many existing implicit models of generative

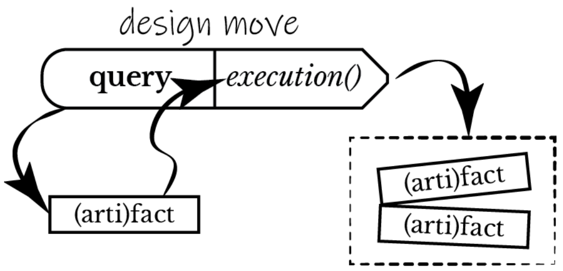


Figure 12.6: The operation of a design move, consisting of a sensory query (which binds data from the blackboard) and the transformation execution function (which operates on the bound data and produces new (arti)facts). The data can be entire generative artifacts or just artifact fragments.

systems operate in such tools closely resembles the ecological design moves (Sec. 4).

Listing 12.4: An example of a design move from a Game Boy ROM generator. This design move adds a background image to a GB Studio scene that doesn't have a background yet, using a collection of known background images. The anonymous :exec function updates the scene's blackboard record to add the image as the new background. The design move transformation functions can be Javascript or Clojurescript, or, as mentioned above, in any other language via calling out to a web server.

```
{:name "add-existing-background-to-scene"
 :comment "Adds an existing background (chosen at random) to a scene that
    ↪ doesn't have a background yet."
 :query  '[:find ?scene ?e ?background-id ?bkg-size
           :in $ %
           :where
           [?scene :scene/uuid ?uuid]
           [(missing? $ ?scene :scene/background)]
           [?e :background/uuid ?background-id]
           [?e :background/size ?bkg-size]]
 :exec
 (fn [db [scene bkg bkg-uuid bkg-size]]
   [{:db/id scene
     :scene/background bkg
     }])}
```

12.4.2 Ecological *Elite*

In addition to the pipeline version of the *Elite* generator, I implemented an ecological version, with design moves for each of the component parts.

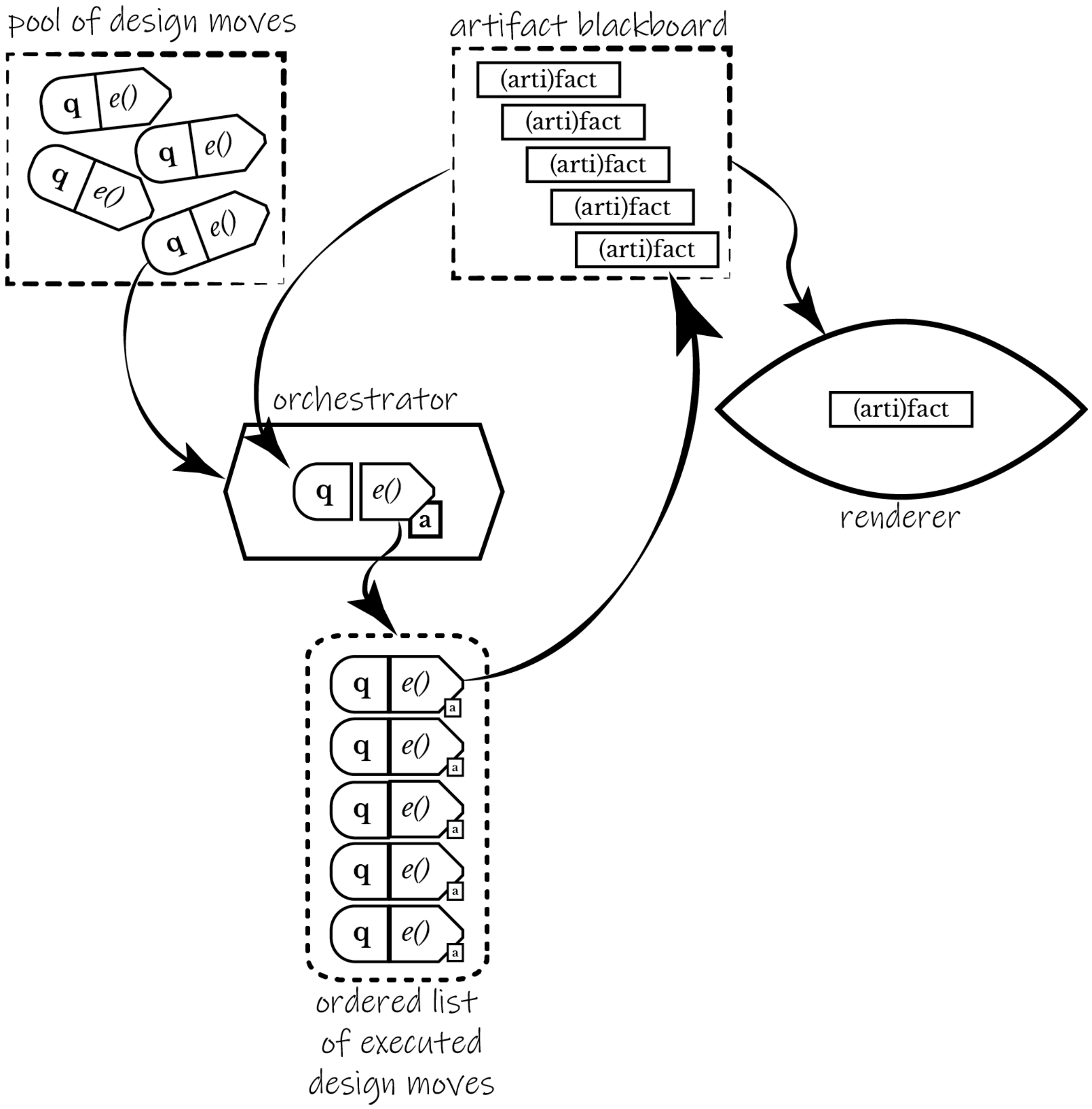


Figure 12.7: One architecture for an Ecological Generative System. The pool of design moves is filtered to the subset that have successful queries with bound (arti)facts. The orchestrator selects the next design move from among them, adding it to the list of executed design moves, executing its translation function, and using the resulting transaction to update the blackboard. The resulting artifacts can be rendered out at any point in the process.

```
{:name "make-planet"
    :exec
    (fn [galaxy-seed galaxy-index]
      (let [new-planet-index (get-new-planet-index elite-db-conn
          ↪ galaxy-index)
            new-planet (make-planet galaxy-seed galaxy-index
                ↪ new-planet-index)]
        new-planet))
    :query-data
    (fn []
      (filter #(< (galaxy-planet-count %)
                  limit-to-galaxy-planet-count)
              (range limit-to-galaxy-count)))
    :query
    '[:find ?galaxy-seed ?galaxy-index
      :in $  [?allowed-galaxy ...]
      :where
      [?galaxy-id :galaxy/seed ?galaxy-seed]
      [?galaxy-id :galaxy/index ?galaxy-index]
      [?planet-id :planet/galaxy ?galaxy-index]
      [?planet-id :planet/index ?planet-index]
      [(= ?galaxy-index ?allowed-galaxy)]
      ]}
```

Listing 12.5: The design move to make a planet. The :query checks the current blackboard environment for the right conditions: having a planet index and galaxy seed; :query-data adds an additional limit on the number of planets per galaxy, though there are a couple of ways that could have been implemented to get the same result. The :exec function invokes the actual generative operation, which returns the data to add to the blackboard: the planet seed, encapsulated in a database transaction.

Naturally, if we only use design moves based on the original generative operations, we produce the same pipeline as the original generator. This is somewhat less efficient, since we're putting the data on the blackboard in pieces, rather than generating it *just-in-time in the middle of the text printing routine* (Sec. 12.1.5). Where the ecological approach is most useful is in meta-generation: constructing a tool to generate a generative system, making live changes to a generative system, or being flexible about the order of generation.

For example, the original *Elite* planet generator has to start with a seed from the exact PRNG sequence: you get whatever planet you end up with, with no later adjustments. To find a good starting planet, the developers had to search until they found a seed that generated a reasonable neighborhood. Well, reasonable by their rather loose standards: once it was released, *Elite* had a reputation as a rather punishing game [124]. Still, the starting area they ended up deciding on has a healthy balance. If we want our generator to start the player on a similar planet with reasonable market prices, nearby trading opportunities, and no pirates, we'd have to do the same—discarding entire galaxies of stars until we found a good seed. A search-based approach or constraint solving would work fine for the relatively small galaxies in *Elite*, but if we had a galaxy the size of *No Man's Sky* or *Elite: Dangerous* that would be impractical.

We could, instead, rewrite the generator to make an exception for the starting planet and override attributes, adding a bunch of special case code that is only used once. Any other exceptions we want to make would require similar edits. This is actually similar to the approach that later versions of *Elite* used to generate special planet descriptions for missions, using some of the valuable slots in the extended text token table to handle these special cases.[21] This is a resource-and-labor-intensive solution, and is likely one of the reasons why only a handful of missions were added.

Or we can use the ecological approach and generalize: we introduce a design move that starts planet generation from one of the attributes instead. The population calculation generative operation doesn't care where the data comes from, after all. Because of the way that *Elite* uses seeds, this requires some re-engineering to run some

[21]As elaborated on at: `https://www.bbcelite.com/deep_dives/extended_system_descriptions.html` [251]

of the seed-dependent operations in reverse, but it gives us more flexibility.

Something we learned from the vivisection of *Elite* is that the planet description is wholly disconnected from the rest of the attributes (Fig. 12.3): no matter how many civil wars and sitcoms are mentioned in the description, they have no direct connection with the rest of the generation. With the original hardware limitations this makes sense; but with our more flexible architecture we can do better. With a pipeline, we have to place the description generation at a fixed point in the pipeline: either before the attributes (with a reading operation to feed data into them) or after them (interpreting the numbers in a way that affects the grammar). If we take an ecological approach we can do either. Or both: generate a description, generate attributes based on that description, revise the description later if the attributes change. We can make decisions about online versus offline on a per-operator basis.

The ecological approach also gives us a method to incorporate AI into the process of designing the generative system: humans aren't the only ones who can select design moves. From another perspective, a design move is a change to a graph expressed as a defined operation. This makes it particularly powerful in mixed-initiative contexts: metaphorically, both participants are speaking the same language.

For me, the real value of the ecological approach is that it flips our perspective on the relation between the nodes and edges: we usually concentrate on the generative operations, because those processes enact the transformations that make the generator happen. But with the ecological approach we can place the data on equal footing: the blackboard lets us look at all of the data in a pile, organized by type but no longer hidden in the links between the outputs and the inputs.

12.5 Conclusion

Looking at *Elite* is valuable, even nearly forty years later, because many of the things it implements are still important considerations for future procgen systems. While our future generators will have different details in their resource tradeoffs, we still need to make decisions about how to create the generator that expresses what we want within the limitations that we face. While we might not need to pack a grammar into 256 tokens, the general pattern of creating correlations between grammar options is powerful. We can extend that to other aspects of the game.

Elite's generator has greater interest than it might otherwise because it leverages different aesthetic effects across time and scale. It manages to give the impression of infinite possibility with only 2048 stars because the way it uses that variation and the rituals you must enact to encounter them maximize the player's perception of the galaxies as living worlds.

12.6 Acknowledgements

I want to thank Jasmine Otto for the visualizations and her assistance with analyzing *Elite* trade routes. I haven't corresponded directly with Mark Moxon, but he should be commended for his immensely useful documentation project.

Chapter 13

Conclusion

This dissertation has explored the idea that all information that emerges from a generator must first be inserted into the generator. Interpreting the compositional parts, we can trace the relationship between inserted input to emergent output.

This dissertation has offered answers to three questions (Fig. 13.1):

- RQ1 (aesthetics): How does the structure of a generative system impact the aesthetic effect of the output?
- RQ2 (composition): What are the lower level components that make up a generative system, and how do those components fit together?
- RQ3 (applications): What new applications of generative systems are unlocked by taking a compositional view of their structure?

Against the background of previous frameworks that attempt to explain procgen (Chap. 3), the vivisection framework (Chap. 8) I have presented here emphasizes the

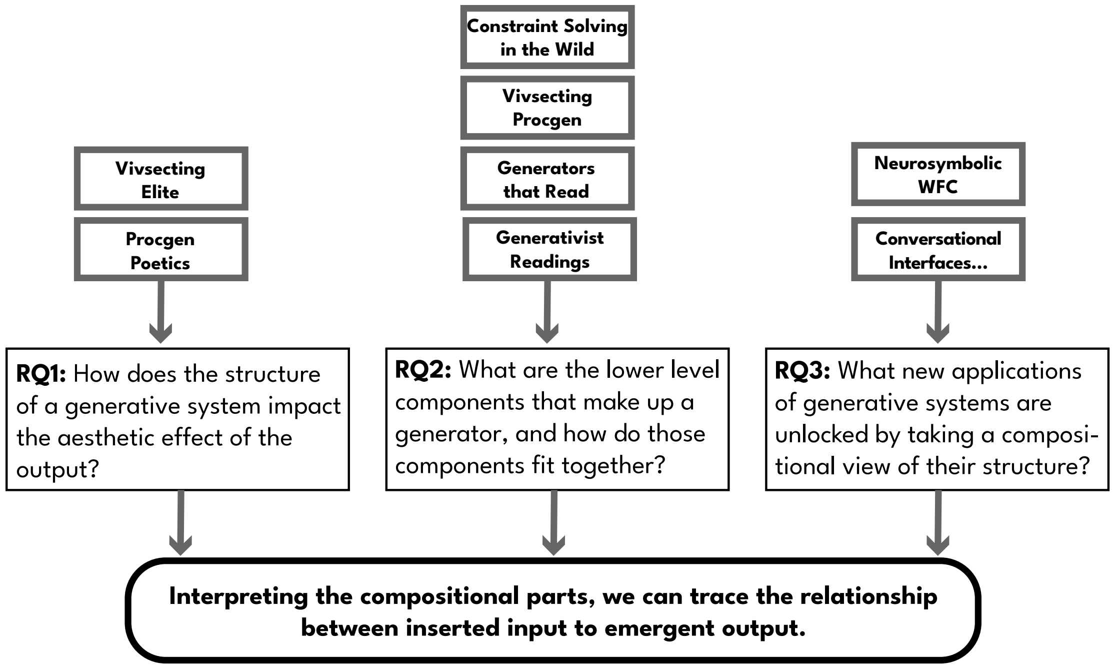

Constraint Solving in the Wild
Vivsecting Procgen
Vivsecting Elite
Generators that Read
Neurosymbolic WFC
Procgen Poetics
Generativist Readings
Conversational Interfaces...
RQ1: How does the structure of a generative system impact the aesthetic effect of the output?
RQ2: What are the lower level components that make up a generator, and how do those components fit together?
RQ3: What new applications of generative systems are unlocked by taking a compositional view of their structure?
Interpreting the compositional parts, we can trace the relationship between inserted input to emergent output.

Figure 13.1: The relationship between the chapters and the research questions, duplicated from Fig. 1.1

modular nature of generative systems and the flow of information within them. This is often reflected in the folk-practices and designs for tools for building generative systems (Chap. 4).

Procgen poetics is about how we perceive the ideas the generative system expresses (Chap. 5). The inclusion of generativity in the creation of an artifact changes the idea that it represents. Equally, comparing two different artifacts from the same system lets us sense the shadow cast by the generative system, giving us a glimpse of the ideas inside.

We can teach generative systems to read, and reading in this sense is an intrinsic part of many generators already (Chap. 6). Performing a generativist reading by interpreting an artifact—via creating the rules that could create it—gives us insight into the system that created it (Chap. 7).

Opening up the generative system and examining it as modular pieces makes it much easier to examine generative systems as they are built and used in practice (Chap. 8). Generative systems *transform data*, potentially from multiple sources at the same time. We can draw graphs of the pipelines that string together these data transformations, classifying the processes by the relation of their input to their output. Beyond that, generative systems also describe *generative possibility spaces*; filtering operations—such as machine learning models or constraint solvers—subdivide and shape the generative possibility space to limit it to a subset of the artifacts, hopefully the subset that we want.

The topology of a generative system is important to its poetics (Sec. 3.10). The process it follows may or may not imitate a the natural creation process of the thing being generated (Sec. 6.2.2.1): generative systems aren't fundamentally about things (Sec. 8.6). They can be used to signify things, but are not the thing that is being signified. The generated artifact is shaped by the idea, but the artifact is not the idea.

The notion that an artist can express an design to a constraint solver via an image drives home the importance of giving artists the ability to easily author expressions of their ideas (Chap. 9). I also demonstrated how to implement the algorithm, making it easier for practitioners to implement their own versions, and suggested some directions for researchers.

My conversational interface project is nominally about WFC, but is also about how you might approach designing the conversation between practitioner and machine in the process of building the generative system (Chap. 10). The fundamental tension discussed in the chapter is the mismatch between the vast need for training data and the small size of the data that exists for a novel context: particularly important for games, which are by definition have novelty at their core. In other words, they need a new idea.

WFC has been primarily looked at as a constraint solver. However, generative systems are modular, so we can upgrade the machine learning model component—demonstrating a generative system that is both constraint solving and PCGML (Chap. 11). This opens up a host of possibilities for combining generative operations that use radically different paradigms.

Finally, the chapter on *Elite* united the perspectives, applying the theoretical models to an actual generative system that has been widely referenced in both the academic literature and by practitioners inspired by it (Chap. 12). *Elite* succeeded because it uses procgen to compress a powerful collection of ideas into a manageable representation.

As we continue toward a world with more generativity [190], with machine learning in every garage and generative models on every street corner, it is all the more necessary to focus on the role that ideas play in procgen. Even fanciest artificial intelligence grinds to a halt without a supply of human ideas to learn from. But at the same time, simple equations can hide dazzlingly complex information. Procgen has been facing more controversy now than ever before. In this chaotic time for procgen and generativity more broadly, it is my hope that this dissertation will serve as a guide to navigating the space ahead.

Bibliography

[1] Cycling '74. *Max*, 6, 2011. URL: `https://cycling74.com/products/max`.

[2] Margarita R Gamarra Acosta, Juan C Vélez Dıaz, and Norelli Schettini Castro. An innovative image-processing model for rust detection using perlin noise to simulate oxide textures. *Corrosion science*, 88:141–151, 2014.

[3] Adobe. *After Effects*, CS6. URL: `https://www.adobe.com/products/aftereffects.html`.

[4] Adobe. *Photoshop*, CS6. URL: `https://www.adobe.com/products/photoshop.html`.

[5] Adobe. Nodes reference, 2020. URL: `https://docs.substance3d.com/sddoc/nodes-reference-129368078.html`.

[6] Adobe. *Substance Designer*, 2020. URL: `https://www.substance3d.com/products/substance-designer/`.

[7] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[8] Alba Amato. Procedural content generation in the game industry. In Oliver Korn and Newton Lee, editors, *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*, pages 15–25. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-53088-8_2.

[9] P. W. Anderson. More is different. *Science*, 177(4047):393–396, 1972. URL: https://www.science.org/doi/abs/10.1126/science.177.4047.393, arXiv:https://www.science.org/doi/pdf/10.1126/science.177.4047.393, doi:10.1126/science.177.4047.393.

[10] Ben. [b...@minimalist.org.uk] Ashmead. Re: TDTTOE and RGRNTOE? (Still learning the acronyms...). [rec.games.roguelike.nethack, 6 March 2004], March 2004. URL: https://groups.google.com/g/rec.games.roguelike.nethack/c/wFFpjR57_9Q/m/BjaCmtB_RIQJ.

[11] Autodesk. Modifier stack right-click menu. URL: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/3DSMax-Basics/files/GUID-3B752392-7B10-4D03-B624-51044BF94106-htm.html.

[12] Autodesk. Transforms, modifiers, and object data flow, 2017. URL: https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2017/ENU/3DSMax/files/GUID-D0820931-3584-49CF-945B-A10BCF9C0814-htm.html.

[13] Autodesk. To use the modifier stack, 2018. URL: `https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/3DSMax-Basics/files/GUID-8209526B-B4EC-406D-A3E2-D43EA717A28A-htm.html`.

[14] Autodesk, Inc. *3ds Max*, 2018. URL: `https://www.autodesk.com/products/3ds-max/overview`.

[15] Mahdi Bahrami. *Engare*, [PC, 2017], 2017.

[16] The Game Band. Blaseball [online]. 2020. URL: `https://www.blaseball.com`.

[17] Alan H. Barr. Teleological modeling. In Norman I. Badler, Brian A. Barsky, and David Zeltzer, editors, *Making Them Move*, pages 315–321. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991. URL: `http://dl.acm.org/citation.cfm?id=111154.111171`.

[18] Nuno Barreto, Amílcar Cardoso, and Licínio Roque. Computational creativity in procedural content generation: A state of the art survey. In *Proceedings of the 2014 conference of science and art of video games*, 2014.

[19] Nicolas A Barriga. A short introduction to procedural content generation algorithms for videogames. *International Journal on Artificial Intelligence Tools*, 28(02):1930001, 2019.

[20] Gabriella Alves Bulhoes Barros, Michael Green, Antonios Liapis, and Julian To-

gelius. Who killed albert einstein? from open data to murder mystery games. *IEEE Transactions on Games*, 2019.

[21] Jonathan Basile. “Library of Babel” [online]. 2015. URL: `https://libraryofbabel.info/`.

[22] Bay 12 Games. *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress*. `http://www.bay12games.com/dwarves/older_versions.html`, [PC, v0.44.05], 2018.

[23] Mark A. Bedau. Weak emergence. *Philosophical Perspectives*, 11:375–399, 1997. URL: `http://www.jstor.org/stable/2216138`.

[24] Ian Bell. txtelite.c, 1999–2015. URL: `http://www.elitehomepage.org/archive/a/b9101315.zip`.

[25] Ian Bell. txtelite.c 1.5, 2015. URL: `http://www.elitehomepage.org/archive/a/b9101315.zip`.

[26] Bethesda Game Studios. *The Elder Scrolls IV: Oblivion*, Windows, 2006.

[27] Blender.org. Modifiers: Introduction, 9 2020. URL: `https://docs.blender.org/manual/en/latest/modeling/modifiers/introduction.html`.

[28] Blinkbat Games. *Desert Golfing*, 2014.

[29] Blizzard North. *Diablo*, 1997.

[30] Philip Bontrager and Julian Togelius. Fully differentiable procedural content generation through generative playing networks, 2020. arXiv:2002.05259.

[31] Michael Booth. The ai systems of left 4 dead. In *Artificial Intelligence and Interactive Digital Entertainment Conference at Stanford, 2009*, 2009. URL: https://steamcdn-a.akamaihd.net/apps/valve/2009/ai_systems_of_l4d_mike_booth.pdf.

[32] Jorge Luis Borges. *The book of sand*. Dutton, New York, 1st ed. edition, 1977.

[33] Jorge Luis Borges. On Exactitude in Science. In *Collected Fictions*. Viking, New York, N.Y., U.S.A., 1998.

[34] Jorge Luis Borges. The Library of Babel. In *Collected Fictions*. Viking, New York, N.Y., U.S.A., 1998.

[35] Jorge Luis Borges. *Ficciones*. Esenciales. Rayo, New York, 1. ed. rayo. edition, 2008.

[36] Amaranth Borsuk, Jesper Juul, and Nick Montfort. The deletionist. https://thedeletionist.com, June 2013.

[37] David Braben. Classic game postmortem - elite. In *Game Developer's Conference 2011*, 2011. URL: https://www.gdcvault.com/play/1014628/Classic-Game-Postmortem.

[38] David Braben and Ian Bell. *Elite*, 1984. [BBC Micro].

[39] Richard P. Brent. On the periods of generalized fibonacci recurrences. *arXiv.org*, 2010. URL: `https://arxiv.org/abs/1004.5439`, `doi:10.48550/ARXIV.1004.5439`.

[40] John Bridgman. The story behind NetHack's long-awaited update–the first since 2003 [online]. April 2016. URL: `https://www.gamasutra.com/view/news/269726/The_story_behind_NetHacks_longawaited_updatethe_first_since_2003.php` [cited 21 January 2018].

[41] Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Trans. Graph.*, 26(3):46–es, 7 2007. `doi:10.1145/1276377.1276435`.

[42] Paul Brink. d4090010.txt, 2014. URL: `http://www.elitehomepage.org/archive/a/d4090010.txt`.

[43] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. *arXiv.org*, September 2018. URL: `http://arxiv.org/abs/1809.11096`, `arXiv:1809.11096`.

[44] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, March 2010. `doi:10.1109/TCIAIG.2010.2041928`.

[45] Charles Brian Bucklew. Roguelike celebration 2019, Oct 2019. URL: `https://www.youtube.com/watch?v=fnFj3dOKcIQ`.

[46] Dave Burraston and Ernest Edmonds. Cellular automata in generative electronic music and sonic art: a historical and technical review. *Digital Creativity*, 16(3):165–185, 2005. `arXiv:https://doi.org/10.1080/14626260500370882`, `doi:10.1080/14626260500370882`.

[47] William S. Burroughs. The cut-up method of Brion Gysin. In Noah Wardrip-Fruin and Nick Montfort, editors, *The New Media Reader*. MIT Press, Cambridge, MA, 2003.

[48] Italo Calvino. Prose and anticombinatorics. In Noah Wardrip-Fruin and Nick Montfort, editors, *The New Media Reader*, pages 183–189. MIT Press, 2003.

[49] Jonathan Campbell and Clark Verbrugge. Exploration in nethack with secret discovery. *IEEE Transactions on Games*, 11(4):363–373, 2019. `doi:10.1109/TG.2018.2861759`.

[50] Harry Caplan. The four senses of scriptural interpretation and the mediaeval theory of preaching. *Speculum*, 4(3):282–290, 1929. URL: `http://www.jstor.org/stable/2849551`.

[51] Laura A. Carlson, Christoph Hölscher, Thomas F. Shipley, and Ruth Conroy Dalton. Getting lost in buildings. *Current Directions in Psychological Science*, 19(5):284–289, October 2010.

[52] C. J. Carr and Zack Zukowski. Generating albums with samplernn to imitate

metal, rock, and punk bands. *CoRR*, abs/1811.06633, 2018. URL: `http://arxiv.org/abs/1811.06633`, `arXiv:1811.06633`.

[53] Francois Chollet. *Deep Learning with Python.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.

[54] M Clayton and A Hashimoto. *Visual Design Fundamentals: A Digital Approach.* Charles River Media / Cengage Learning, Boston, MA, 2009.

[55] Walter Cohen. Color-perception in the chromatic Ganzfeld. *The American Journal of Psychology*, 71(2):390–394, June 1958. URL: `http://www.jstor.org/stable/1420084` [cited 28 January 2018].

[56] Simon Colton, John Charnley, and Alison Pease. Computational creativity theory: The face and idea descriptive models. In *Proceedings of the 2nd International Conference on Computational Creativity, ICCC 2011*, Proceedings of the 2nd International Conference on Computational Creativity, ICCC 2011, pages 90–95, December 2011. International Conference on Computational Creativity 2011, ICCC 2011 ; Conference date: 27-04-2011 Through 29-04-2011. URL: `https://computationalcreativity.net/iccc2011/proceedings/index.html`.

[57] Kate Compton. So you want to build a generator [online]. 2016. URL: `http://www.galaxykate.com/blog/generator.html` [cited February 1 2017].

[58] Kate Compton. Idle hands, 2017. URL: `http://www.galaxykate.com/apps/idlehands/`.

[59] Kate Compton. *Casual Creators Defining a Genre of Autotelic Creativity Support Systems*. PhD thesis, UC Santa Cruz, 2019. URL: `https://escholarship.org/uc/item/4kg8g9gd`.

[60] Kate Compton. personal correspondence, August 2022.

[61] Kate Compton. personal correspondence, July 2022.

[62] Kate Compton. personal correspondence, October 2022.

[63] Kate Compton, Ben Kybartas, and Michael Mateas. Tracery: An author-focused generative text tool. In Henrik Schoenau-Fog, Luis Emilio Bruni, Sandy Louchart, and Sarune Baceviciute, editors, *Interactive Storytelling*, pages 154–161, Cham, 2015. Springer International Publishing.

[64] Kate Compton and Michael Mateas. A generative framework of generativity. In *Experimental AI in Games Workshop 2017, at the Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, Snowbird, Little Cottonwood Canyon, Utah USA, October 2017. The AAAI Press, Palo Alto, California. URL: `https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15896`.

[65] Kate Compton, Edward Melcer, and Michael Mateas. Generominos: Ideation cards for interactive generativity. In *Experimental AI in Games Workshop 2017, at the Thirteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Snowbird, Little Cottonwood Canyon, Utah USA, 2017. AAAI

Press. URL: https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15898.

[66] Kate Compton, Joseph C Osborn, and Michael Mateas. Generative methods. In *The Fourth Procedural Content Generation in Games workshop*, 2013. URL: http://www.fdg2013.org/program/workshops/papers/PCG2013/pcg2013_6.pdf.

[67] Kate Compton, Johnathan Pagnutti, and Jim Whitehead. A shared language for creative communities of artbots. In *Proceedings of the 2017 Co-Creation Workshop*, Atlanta, Georgia, USA, June 2017. Eighth International Conference on Computational Creativity.

[68] Kieran Connell. elite-beebasm, 2018. URL: https://github.com/kieranhj/elite-beebasm.

[69] Michael Connor. Why is Deep Dream turning the world into a doggy monster hellscape? [online]. July 2015. URL: http://rhizome.org/editorial/2015/jul/10/deep-dream-doggy-monster/ [cited 2 December 2018].

[70] NaNoGenMo contributors. NaNoGenMo [online]. Nov 2021. URL: https://nanogenmo.github.io/.

[71] Matthew Cook et al. Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.

[72] Michael Cook. Make Something That Makes Something: A Report On The First

Procedural Generation Jam. In *Proceedings of the Sixth International Conference on Computational Creativity*, pages 197–203, 2015.

[73] Michael Cook. Alien languages: How we talk about procedural generation [online]. August 2016. URL: `https://www.gamedeveloper.com/design/alien-languages-how-we-talk-about-procedural-generation` [cited 9 July 2022].

[74] Michael Cook and Simon Colton. Redesigning computationally creative systems for continuous creation. In Francois Pachet, Anna Jordanous, and Leon Carlos, editors, *Proceedings of the Ninth International Conference on Computational Creativity, ICCC 2018, Salamanca, Spain, June 25-29, 2018.*, pages 32–39. Association for Computational Creativity (ACC), Online, July 2018. URL: `http://repository.falmouth.ac.uk/2943/`.

[75] Michael Cook, Simon Colton, and Jeremy Gow. The ANGELINA videogame design system—part II. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(3):254–266, 2016.

[76] Michael Cook, Jeremy Gow, Gillian Smith, and Simon Colton. Danesh: Interactive tools for understanding procedural content generators. *IEEE Transactions on Games*, 2021. `doi:10.1109/TG.2021.3078323`.

[77] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. URL: `http://www.jstor.org/stable/2003354`.

[78] Seth Cooper, Firas Khatib, Adrien Treuille, Janos Barbero, Jeehyung Lee, Michael Beenen, Andrew Leaver-Fay, David Baker, Zoran Popović, and Foldit players. Predicting protein structures with a multiplayer online game. *Nature*, 08 2010. doi:10.1038/nature09304.

[79] Jonathan Culler. *Literary Theory: A Very Short Introduction*. Oxford University Press, 1997.

[80] Jonathan Culler. *Theory of the Lyric*, page 6. Harvard University Press, Cambridge, MA, 2015.

[81] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., GBR, 1972.

[82] James Davenport. From the Elder Scrolls to the US Secret Service: Where videogame trees come from [online]. September 2017. URL: http://www.pcgamer.com/from-the-elder-scrolls-to-the-us-secret-service-where-videogame-trees-come-from/ [cited 12 December 2017].

[83] Rafael Guerra de Pontes and Herman Martins Gomes. Evolutionary procedural content generation for an endless platform game. In *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 80–89. IEEE, 2020.

[84] J. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 11 1980. doi:10.1109/MC.1980.1653418.

[85] Bruno Dias. Procedural meaning: Pragmatic procgen in voyageur [online]. 2016. URL: `https://www.gamasutra.com/blogs/BrunoDias/20160718/277314/Procedural_meaning_Pragmatic_procgen_in_Voyageur.php`.

[86] Bruno Dias. "So, lesson learned with @VoyageurGame and #procgen. Voyageur uses a "recombinant" system to make planet descs, right? it assembles them...." [online]. May 2017. URL: `https://twitter.com/NotBrunoAgain/status/867098020596285442` [cited August 2018].

[87] Bruno Dias. *Voyageur*, [Android], 2017.

[88] Edsger W. Dijkstra. On the economy of doing mathematics. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, pages 2–10, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[89] Alan Dorin, Jonathan McCabe, Jon McCormack, Gordon Monro, and Mitchell Whitelaw. A framework for understanding generative art. *Digital Creativity*, 23(3-4):239–259, 2012. `arXiv:https://doi.org/10.1080/14626268.2012.709940`, `doi:10.1080/14626268.2012.709940`.

[90] Joris Dormans. *Engineering emergence: applied theory for game design.* PhD thesis, University of Amsterdam, Netherlands, 2012.

[91] Joris Dormans. Cyclic generation. In *Procedural Generation in Game Design*, pages 83–96. AK Peters/CRC Press, 2017.

[92] Angus J. C. Duggan. Elite A, late 1980s. URL: `http://knackered.org/angus/beeb/elite.html`.

[93] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, apr 2020. `doi:10.1109/tetc.2017.2785299`.

[94] Df2014:engraving [online]. October 2018. URL: `http://dwarffortresswiki.org/index.php?title=DF2014:Engraving&oldid=237992` [cited 2 December 2018].

[95] David Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach (Third Edition)*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, San Francisco, third edition edition, 2003.

[96] David S Ebert, Randall M Rohrer, Christopher D Shaw, Pradyut Panda, James M Kukla, and D.Aaron Roberts. Procedural shape generation for multi-dimensional data visualization. *Computers & Graphics*, 24(3):375–384, 2000. URL: `https://www.sciencedirect.com/science/article/pii/S0097849300000339`, `doi:https://doi.org/10.1016/S0097-8493(00)00033-9`.

[97] Umberto Eco. Make your own movie. In *Misreadings*, pages 145–155. Harcourt Brace & Co., San Diego, 1993.

[98] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and

transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 341–346, New York, NY, USA, 2001. ACM. URL: `http://doi.acm.org/10.1145/383259.383296`, `doi:10.1145/383259.383296`.

[99] Alexei A Efros and Thomas K Leung. Texture synthesis by non-parametric sampling. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1033–1038. IEEE, IEEE Computer Society, 1999, 1999.

[100] Arnaud Emilien, Adrien Bernhardt, Adrien Peytavie, Marie-Paule Cani, and Eric Galin. Procedural generation of villages on arbitrary terrains. *The Visual Computer*, 28(6-8):809–818, 2012.

[101] Brian Eno. *A Year With Swollen Appendices.* Faber and Faber Ltd., 3 Queen Square, London, UK, 1996.

[102] Ensemble Studios. *Age of Empires II: The Age of Kings*, Windows, 1999.

[103] Ensemble Studios and Hidden Path Entertainment. *Age of Empires II: The Age of Kings: HD Edition*, Windows, 2013.

[104] Patrick Esser, Robin Rombach, and Bjorn Ommer. Taming transformers for high-resolution image synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12873–12883, June 2021.

[105] Andrea Falcon. Aristotle on Causality. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2022 edition, 2022.

[106] Mathieu Fehr and Nathanaël Courant. fast-wfc. *GitHub repository*, 2018.

[107] P. Ffrench. Text. In Roland Greene, Stephen Cushman, Clare Cavanagh, Jahan Ramazani, Paul F. Rouzer, Harris Feinsod, David Marno, Alexandra Slessarev, and Inc. ebrary, editors, *The Princeton Encyclopedia of Poetry and Poetics : Fourth Edition*, pages 1425–1426. Princeton University Press, 41 William Street, Princeton, New Jersey 08540, 2012.

[108] Leif Foged and Ian D Horswill. *Rolling Your Own Finite-Domain Constraint Solver*, pages 283–302. A K Peters/CRC Press, 2015.

[109] Amanda L. French. *Refrain, again: The return of the villanelle*. PhD thesis, University of Virginia, 2004. URL: `https://villanelle.amandafrench.net/`.

[110] AMANDA L. FRENCH. Edmund gosse and the stubborn villanelle blunder. *Victorian Poetry*, 48(2):243–266, 2010. URL: `http://www.jstor.org/stable/27896675`.

[111] Jochen Fromm. Types and forms of emergence, 2005. URL: `https://arxiv.org/abs/nlin/0506028`, `doi:10.48550/ARXIV.NLIN/0506028`.

[112] Frontier Developments. *Elite: Dangerous*, 1.0, 2014.

[113] Philip Galanter. What is generative art? complexity theory as a context for art theory. In *In GA2003 – 6th Generative Art Conference*, 2003.

[114] Philip Galanter. *Generative Art Theory*, chapter 5, pages 146–180. John Wiley & Sons, Ltd, 2016. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118475249.ch5`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118475249.ch5`, `doi:https://doi.org/10.1002/9781118475249.ch5`.

[115] Special edition podcast: No Man's Sky [online]. 2014. URL: `http://www.gameinformer.com/cfs-filesystemfile.ashx/__key/CommunityServer-Components-SiteFiles/media-audio-theshow-nomanssky/nmsspecialedition.mp3`.

[116] Brace Yourself Games. *Crypt of the Necrodancer*, 2015.

[117] People Make Games. Why the sound of a gun had to be nerfed in wolfenstein: Enemy territory, August 2019. URL: `https://www.youtube.com/watch?v=RDxiuHdR_T4`.

[118] David Donovan Garber. *Computational Models for Texture Analysis and Texture Synthesis*. PhD thesis, University of Southern California, Los Angeles, CA, USA, 1981. AAI0551115.

[119] Artur D'avila Garcez and Luis C Lamb. Neurosymbolic AI: The 3rd wave. *arXiv.org*, December 2020. URL: `http://arxiv.org/abs/2012.05876`, `arXiv:2012.05876`.

[120] Sofia Garcia. Helena Sarin: Celebrating the original GANcomic, October 2020. URL: `https://www.artxcode.io/journal/helena-sarin-celebrating-the-original-gancomic`.

[121] Martin Gardner. The fantastic combinations of john conway's new solitaire game 'life'. *Scientific American*, 233(4):120–123, October 1970.

[122] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015. URL: `http://arxiv.org/abs/1508.06576`.

[123] Phoebe Gavin. The infinite world of no man's sky: How a game created 18 quintillion planets to explore, 7 2015. URL: `https://slate.com/human-interest/2015/07/no-mans-sky-infinite-video-game-universe-created-with-algorithms-video.html`.

[124] Alison Gazzard. The platform and the player: exploring the (hi) stories of elite. *Game Studies*, 13(2), 2013. URL: `http://gamestudies.org/1302/articles/agazzard`.

[125] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[126] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.

[127] Alessandro Ghignola. *Noctis IV*, 2001.

[128] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[129] Gaël Glorian, Adrien Debesson, Sylvain Yvon-Paliot, and Laurent Simon. The Dungeon Variations Problem Using Constraint Programming. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/15318`, `doi:10.4230/LIPIcs.CP.2021.27`.

[130] AI Religion Bot @gods_txt. We may hope to make all things new through the creativity that is left, but in a world of chance there will always be an element of chance. Thus, while we may seek after the eternal and the holy, the temporal and profane will forever remain chained to its form., 2 2023. URL: `https://twitter.com/gods_txt/status/1622616854777982977`.

[131] Carla P Gomes, Ashish Sabharwal, and Bart Selman. Near-uniform sampling of combinatorial spaces using xor constraints. In *Advances in Neural Information Processing Systems*, pages 481–488, 2006.

[132] Frank Granger, editor. *Vitruvius on Architecture*, volume 1. William Heinemann Ltd, London, 1931.

[133] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Procedural content generation through quality diversity. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[134] Bert F. Green, J. E. Keith Smith, and Laura Klem. Empirical tests of an additive random number generator. *J. ACM*, 6(4):527–537, oct 1959. doi:10.1145/320998.321006.

[135] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Undiscovered worlds–towards a framework for real-time procedural world generation. In *Fifth International Digital Arts and Culture Conference, Melbourne, Australia*, volume 5, page 5, 2003.

[136] Jason Grinblat. Generating histories. In *Procedural Storytelling in Game Design*, pages 179–192. AK Peters/CRC Press, 2019.

[137] Jason Grinblat and Brian Bucklew. Math for game developers: End-to-end procedural generation in 'caves of qud'. In *Game Developer's Conference 2019*, San Francisco, CA USA, March 2019.

[138] Jason Grinblat and C. Brian Bucklew. Subverting historical cause & effect: Generation of mythic biographies in caves of qud. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3102071.3110574.

[139] Jason Grinblat and Charles B. Bucklew. Caves of Qud, 2010.

[140] Maxim Gumin. Bitmap & tilemap generation from a single example by collapsing a wave function https://github.com/mxgmn/wavefunctioncollapse, Sep 2016. URL: `https://twitter.com/ExUtumno/status/781834584136814593`.

[141] Maxim Gumin. Convchain. *GitHub repository*, 2016.

[142] Maxim Gumin. Syntex. *GitHub repository*, 2016.

[143] Maxim Gumin. WaveFunctionCollapse. *GitHub repository*, 2016. URL: `https://github.com/mxgmn/WaveFunctionCollapse`.

[144] Maxim Gumin. WaveFunctionCollapse. *GitHub repository*, 2016.

[145] Maxim Gumin. WaveFunctionCollapse Readme.md, May 2017. URL: `https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md`.

[146] Matthew Guzdial, Nicholas Liao, Jonathan Chen, Shao-Yu Chen, Shukan Shah, Vishwa Shah, Joshua Reno, Gillian Smith, and Mark Riedl. Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators. In *Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 2019.

[147] Matthew Guzdial, Nicholas Liao, and Mark Riedl. Co-creative level design via machine learning. In *The 5th Experimental AI in Games Workshop (EXAG)*, 2018. URL: `http://ceur-ws.org/Vol-2282/EXAG_126.pdf`.

[148] Matthew Guzdial, Joshua Reno, Jonathan Chen, Gillian Smith, and Mark Riedl. Explainable pcgml via game design patterns. In *The 5th Experimental AI in Games Workshop (EXAG)*, 2018. URL: `http://ceur-ws.org/Vol-2282/EXAG_107.pdf`.

[149] Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346, 1990.

[150] Paul Francis Harrison. *Image Texture Tools: Texture Synthesis, Texture Transfer, and Plausible Restoration.* Monash University, 2006.

[151] Erin Jonathan Hastings, Ratan K Guha, and Kenneth O Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, 2009.

[152] Stephen A. Hedges. Dice music in the eighteenth century. *Music & Letters*, 59(2):180–87, 1978.

[153] Hello Games. *No Man's Sky*, Windows, 1.0, 2016.

[154] Linley Henzell, Brent Ross, and The Dungeon Crawl Stone Soup team. *Dungeon Crawl Stone Soup*, 0.28.0, 2022. URL: `https://crawl.develz.org/`.

[155] Aaron Hertzmann, Charles E Jacobs, Nuria Oliver, Brian Curless, and David H Salesin. Image analogies. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340. ACM, 2001.

[156] Chris Higgins. No Man's Sky would take 5 billion years to explore [online]. August 2014. URL: `https://www.wired.co.uk/article/no-mans-sky-planets`.

[157] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. `arXiv:https://doi.org/10.1162/neco.1997.9.8.1735`, `doi:10.1162/neco.1997.9.8.1735`.

[158] R. Hollander. *Dante: A Life in Works.* Yale University Press, 2001.

[159] Taylor Holmes. Interview with phenomenal game designer oskar stälberg, Jan 2016. URL: `https://taylorholmes.com/2016/01/22/interview-with-phenomenal-game-designer-oskar-stalberg/`.

[160] Ian Horswill. Imaginarium: A tool for casual constraint-based pcg. In *Proceedings of the AIIDE Workshop on Experimental AI and Games (EXAG)*, 2019.

[161] Ian D Horswill and Leif Foged. Fast procedural level population with playability constraints. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[162] Caleb Howard. Asking the impossible on ssx: Creating 300 tracks on a ten-track budget. URL: `https://www.gdcvault.com/play/1015547/Asking-the-Impossible-on-SSX`.

[163] Kathryn Hume. Nanogenmo: Dada 2.0 [online]. 2015. URL: `http://arcade.stanford.edu/blogs/nanogenmo-dada-20` [cited 27 January 2018].

[164] Robin Hunicke. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 429–433, 2005.

[165] Robin Hunicke, Marc LeBlanc, and Robert Zubek. MDA: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4, page 1722, 2004.

[166] IDV inc. Geometry forces [speedtree documentation], January 2022. URL: `https://docs9.speedtree.com/modeler/doku.php?id=geometry_forces`.

[167] Jon Ingold. Designing for narrative momentum. In *Procedural Storytelling in Game Design*, pages 75–89. AK Peters/CRC Press, 2019.

[168] Peter Irving and Jeremy Smith. *Exile*, 1988. [BBC Micro].

[169] Alex Jaffe. Cursed problems in game design, 2019. URL: `https://www.youtube.com/watch?v=8uE6-vIi1rQ`.

[170] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC Workshop on Computational Creativity and Games*, 2016.

[171] Stefan Jänicke, Greta Franzini, Muhammad Faisal Cheema, and Gerik Scheuermann. On close and distant reading in digital humanities: A survey and future challenges. In *Eurographics Conference on Visualization (EuroVis)-STARs. The*

Eurographics Association, page 6, 2015. URL: https://dx.doi.org/10.2312/eurovisstar.20151113, doi:10.2312/eurovisstar.20151113.

[172] Mark Johnson. *Ultima Ratio Regum*, 0.9.0b, January 2022. URL: https://www.markrjohnsongames.com/games/ultima-ratio-regum/.

[173] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

[174] Jesper Juul. The open and the closed: Games of emergence and games of progression. In *Computer Games and Digital Cultures Conference Proceedings*. Tampere University Press, June 2002. URL: http://www.digra.org/wp-content/uploads/digital-library/05164.10096.pdf.

[175] Roland Kaminski, Torsten Schaub, and Philipp Wanko. A tutorial on hybrid answer set solving with clingo, 2017. URL: https://www.cs.uni-potsdam.de/~torsten/hybris.pdf.

[176] Frank Kane. *Modeling, Lighting, and Rendering Techniques for Volumetric Clouds*, pages 21–43. AK Peters, Ltd., 2011.

[177] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018. URL: http://arxiv.org/abs/1812.04948, arXiv:1812.04948.

[178] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. In *Proceedings*

of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 8110–8119. openaccess.thecvf.com, 2020. URL: `http://openaccess.thecvf.com/content_CVPR_2020/html/Karras_Analyzing_and_Improving_the_Image_Quality_of_StyleGAN_CVPR_2020_paper.html`.

[179] Isaac Karth. Ergodic agency: How play manifests understanding. In Dawn Stobbart and Monica Evans, editors, *Engaging with Videogames: Play, Theory and Practice*, pages 205–216. Inter-Disciplinary Press, 2014.

[180] Isaac Karth. Elite: Dangerous - stations [online]. 2015. URL: `https://procedural-generation.tumblr.com/post/134545517646/elite-dangerous-stations-something-thats-a-bit` [cited 2 December 2018)].

[181] Isaac Karth. *Virgil's Commonplace Book*. `https://github.com/ikarth/ViaAppiaNovel/raw/master/via_appia_nanogenmo.pdf`, November 2015. URL: `https://github.com/ikarth/ViaAppiaNovel`.

[182] Isaac Karth. Preliminary poetics of procedural generation in games. In *DiGRA '18 - Proceedings of the 2018 DiGRA International Conference: The Game is the Message*. DiGRA, 7 2018. URL: `http://www.digra.org/wp-content/uploads/digital-library/DIGRA_2018_paper_166.pdf`.

[183] Isaac Karth. Preliminary poetics of procedural generation in games. *Transactions of the Digital Games Research Association*, 4(3), 2019. `doi:https://doi.org/10.26503/todigra.v4i3.106`.

[184] Isaac Karth, Batu Aytemiz, Ross Mawhorter, and Adam M. Smith. Neurosymbolic map generation with VQ-VAE and WFC. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021*, FDG'21, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3472538.3472584`.

[185] Isaac Karth, Nic Junius, and Max Kreminski. Constructing a catbox: Story volume poetics in Umineko no Naku Koro ni. In Mirjam Vosmeer and Lissa Holloway-Attaway, editors, *Interactive Storytelling*, pages 455–470, Cham, 12 2022. Springer International Publishing.

[186] Isaac Karth and Adam M. Smith. WaveFunctionCollapse is Constraint Solving in the Wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 14-17 August, FDG '17, pages 68:1–68:10, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3102071.3110566`, `doi:10.1145/3102071.3110566`.

[187] Isaac Karth and Adam M. Smith. Addressing the fundamental tension of PCGML with discriminative learning. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, pages 89:1–89:9, New York, NY, USA, 2019. ACM. URL: `http://doi.acm.org/10.1145/3337722.3341845`, `doi:10.1145/3337722.3341845`.

[188] Isaac Karth and Adam Marshall Smith. Wavefunctioncollapse: Content generation via constraint solving and machine learning. *IEEE Transactions on Games*, pages 1–1, 2021. `doi:10.1109/TG.2021.3076368`.

[189] Karth Karth, Tamara Duplantis, Max Kreminski, Sachita Kashyap, Vijaya Kukutla, Anika Mittal, Harvin Park, and Adam M. Smith. Generating playable rpg roms for the game boy. In *Games and Demonstrations at Foundations of Digital Games 2021*, 2021. URL: `https://escholarship.org/uc/item/5c4068hq`.

[190] Kevin Kelly. Picture limitless creativity at your fingertips: Artificial intelligence can now make better art than most humans. soon, these engines of wow will transform how we design just about everything., November 2022. URL: `https://www.wired.com/story/picture-limitless-creativity-ai-image-generators/`.

[191] Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius, and Georgios N. Yannakakis. A procedural procedural level generator generator. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 335–341, 2012. `doi:10.1109/CIG.2012.6374174`.

[192] Rilla Khaled, Mark J. Nelson, and Pippin Barr. Design metaphors for procedural content generation in games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1509–1518. ACM, 2013. `doi:10.1145/2470654.2466201`.

[193] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1):95–

101, Oct. 2020. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/7416, doi:10.1609/aiide.v16i1.7416.

[194] Matvey Khokhlov, Immanuel Koh, and Jeffrey Huang. Voxel synthesis for generative design. In John S. Gero, editor, *Design Computing and Cognition '18*, pages 227–244, Cham, 2019. Springer International Publishing.

[195] Denis Kozlov. Procedural Content Creation F.A.Q. - Project Aero, Houdini and Beyond, 2017. URL: https://www.the-working-man.org/2017/04/procedural-content-creation-faq-project.html.

[196] Max Kreminski. blackout. https://mkremins.github.io/blackout, March 2017.

[197] Max Kreminski. personal communication, 12 2020.

[198] Max Kreminski and Isaac Karth. A demonstration of blabrecs, an ai-based wordgame. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE) 2021*, 2021. URL: https://dl.acm.org/doi/abs/10.5555/3505520.3505554.

[199] Max Kreminski, Isaac Karth, Michael Mateas, and Noah Wardrip-Fruin. Evaluating mixed-initiative creative interfaces via expressive range coverage analysis. In *HAI-GEN 2022: 3rd Workshop on Human-AI Co-Creation with Generative Models*, 2022.

[200] Max Kreminski, Isaac Karth, and Noah Wardrip-Fruin. Generators that read. In

Proceedings of the 14th International Conference on the Foundations of Digital Games, pages 1–7. ACM, 2019.

[201] Max Kreminski and Noah Wardrip-Fruin. Gardening games: an alternative philosophy of pcg in games. In *The 9th Workshop on Procedural Content Generation (PCG2018)*, PCG2018, 2018. URL: `https://mkremins.github.io/publications/GardeningGames.pdf`.

[202] L.V Kuleshov. *Kuleshov on Film: Writings.* University of California Press, Berkeley and Los Angeles, CA, 1974. Translated by Ronald Levaco.

[203] Rebecca Levene and Magnus Anderson. *Grand Thieves & Tomb Raiders: How British Video Games Conquered the World.* Aurum Press, 2012.

[204] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics (ToG)*, 20(3):127–150, 2001.

[205] A. Liapis, G. N. Yannakakis, and J. Togelius. Designer modeling for sentient sketchbook. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, Aug 2014. `doi:10.1109/CIG.2014.6932873`.

[206] Antonios Liapis, Gillian Smith, and Noor Shaker. Mixed-initiative content creation. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 195–216. Springer, 2016.

[207] Antonios Liapis, Georgios N. Yannakakis, Mark J. Nelson, Mike Preuss, and Rafael Bidarra. Orchestrating game generation. *IEEE Transactions on Games*, 11(1):48–68, 2019. `doi:10.1109/TG.2018.2870876`.

[208] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: Computer-aided game level authoring. In *FDG*, pages 213–220, 2013.

[209] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N Yannakakis, and Julian Togelius. Deep learning for procedural content generation. *Neural Computing and Applications*, 33(1):19–37, January 2021. URL: `https://doi.org/10.1007/s00521-020-05383-8`.

[210] Simon Liu, Li Chaoran, Li Yue, Ma Heng, Hou Xiao, Shen Yiming, Wang Licong, Chen Ze, Guo Xianghao, Lu Hengtong, et al. Automatic generation of tower defense levels using pcg. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–9, 2019.

[211] Yang Liu, Eunice Jun, Qisheng Li, and Jeffrey Heer. Latent space cartography: Visual analysis of vector space embeddings. *Computer Graphics Forum*, 38(3), 2019. URL: `https://par.nsf.gov/biblio/10172008`, `doi:10.1111/cgf.13672`.

[212] Gearbox Software LLC. *Borderlands*, 2009.

[213] Mossmouth LLC. *Spelunky*, 2013.

[214] E.N. Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[215] Yuri Lotman. *The Structure of the Artistic Text.* University of Michigan: Department of Slavic Languages and Literature, Ann Arbor, Michigan, 1977.

[216] Andrew Lowell. Simulation & proceduralism beyond feature film & fx, 2016. URL: `https://www.youtube.com/watch?v=71QsZjU-IZo`.

[217] Simon M Lucas and Vanessa Volz. Tile pattern kl-divergence for analysing and evolving game levels. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 170–178, 2019.

[218] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1 (HLT '11)*, volume 1, pages 142–150, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[219] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* W.H. Freeman, San Francisco, CA, 1983.

[220] Matheson Marcault. The History of Text Generation, 2015. URL: `http://mathesonmarcault.com/index.php/2015/12/15/randomly-generated-title-goes-here/`.

[221] George Marsaglia. Random number generators. *Journal of Modern Applied Statistical Methods*, 2(1):2, 2003. `doi:10.22237/jmasm/1051747320`.

[222] Chris Martens, Adam Summerville, Michael Mateas, Joseph Osborn, Sarah Harmon, Noah Wardrip-Fruin, and Arnav Jhala. Proceduralist readings, procedurally. In *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2016. URL: `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/view/14061`.

[223] Kevin Martens. private communication, October 2018.

[224] S. Maschwitz. *The DV Rebel's Guide: An All-Digital Approach to Making Killer Action Movies on the Cheap*. Pearson Education, 2006. URL: `https://books.google.com/books?id=iha0yoFsSBkC`.

[225] Maxis. *Spore*, 2008.

[226] Stella Mazeika. *Hierarchical Style Modeling: A generative framework for Style-Centric Generation of 3D Models*. PhD thesis, UC Santa Cruz, 2019. URL: `https://escholarship.org/uc/item/4sh5827q`.

[227] Scott McCloud. *Understanding comics: The invisible art*. Tundra, Northampton, MA, 1993.

[228] Brian P. McLaughlin. The Rise and Fall of British Emergentism. In *Emergence: Contemporary Readings in Philosophy and Science*. The MIT Press, 03 2008. `arXiv:https://academic.oup.com/mit-press-scholarship-online/book/0/chapter/167395262/chapter-ag-pdf/44898460/book\_13765\_section\_167395262.ag.pdf`, `doi:10.7551/mitpress/9780262026215.003.0003`.

[229] P. Merrell and D. Manocha. Model synthesis: A general procedural modeling algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 17(6):715–728, June 2011. `doi:10.1109/TVCG.2010.112`.

[230] Paul C Merrell. *Model synthesis*. PhD thesis, University of North Carolina at Chapel Hill, 2009.

[231] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013. URL: `https://arxiv.org/abs/1301.3781`, `doi:10.48550/ARXIV.1301.3781`.

[232] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013. arXiv:1301.3781.

[233] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc. URL: `http://dl.acm.org/citation.cfm?id=2999792.2999959`.

[234] Tomas Mikolov, Wen tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational

Linguistics, May 2013. URL: https://www.microsoft.com/en-us/research/publication/linguistic-regularities-in-continuous-space-word-representations/.

[235] Minecraft Wiki contributors. Altitude, Nov 2018. URL: https://minecraft.fandom.com/wiki/Altitude?oldid=1288263.

[236] Minecraft Wiki contributors. Tutorials/diamonds [online]. November 2018. URL: https://minecraft.gamepedia.com/index.php?title=Tutorials/Diamonds&oldid=1283257 [cited 2 December 2018].

[237] Alex Mitchell, Liting Kway, and Brandon Junhui Lee. Storygameness: understanding repeat experience and the desire for closure in storygames. In *DiGRA 2020–Proceedings of the 2020 DiGRA International Conference*. DiGRA, 2020.

[238] Mojang AB. *Minecraft [Windows Java Version 1.0]*, 2011.

[239] Nick Montfort. *Megawatt*. nickm.com, 2014. URL: https://nickm.com/post/2014/11/megawatt/.

[240] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks, 2015. URL: https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

[241] F. Moretti and A. Piazza. *Graphs, Maps, Trees: Abstract Models for a Literary History*. Verso, 2005.

[242] Mauro Mosconi and Marco Porta. Iteration constructs in data-flow visual programming languages. *Computer languages*, 26(2-4):67–104, 2000.

[243] M. Moxon, I. Bell, and D. Brabin. TKN1 [6502SP version], 1984, 2022. URL: `https://www.bbcelite.com/6502sp/main/variable/tkn1.html`.

[244] Mark Moxon. Drawng explosion clouds, 2020–2022. URL: `https://www.bbcelite.com/deep_dives/drawing_explosion_clouds.html`.

[245] Mark Moxon. Elite on the BBC Micro, 2020–2022. URL: `http://bbelite.com`.

[246] Mark Moxon. Extended text tokens, 2020–2022. URL: `https://www.bbcelite.com/deep_dives/extended_text_tokens.html`.

[247] Mark Moxon. Galaxy and system seeds, 2020–2022. URL: `https://www.bbcelite.com/deep_dives/galaxy_and_system_seeds.html`.

[248] Mark Moxon. Generating system data, 2020–2022. URL: `https://www.bbcelite.com/deep_dives/generating_system_data.html`.

[249] Mark Moxon. Twisting the system seeds, 2020–2022. URL: `https://www.bbcelite.com/deep_dives/twisting_the_system_seeds.html`.

[250] Mark Moxon. About Elite-A: Information on Angus Duggan's extended version of Elite, 2021. URL: `https://www.bbcelite.com/elite-a/`.

[251] Mark Moxon. Extended system descriptions, 2022. URL: `https://www.bbcelite.com/deep_dives/extended_system_descriptions.html`.

[252] Mark Moxon. Fully documented source code for Elite on the BBC Micro, 2022. URL: `https://www.bbcelite.com/`.

[253] K. R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*, 12(2):181–201, Mar 2001. `doi:10.1109/72.914517`.

[254] F. Kenton Musgrave. 14 - a brief introduction to fractals. In David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, William R. Mark, John C. Hart, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley, editors, *Texturing and Modeling (Third Edition)*, The Morgan Kaufmann Series in Computer Graphics, pages 428 – 445. Morgan Kaufmann, San Francisco, third edition edition, 2003. URL: `http://www.sciencedirect.com/science/article/pii/B9781558608481500437`, `doi:https://doi.org/10.1016/B978-155860848-1/50043-7`.

[255] The NetHack devteam. *NetHack*, 3.6.6, 2020. URL: `https://www.nethack.org/common/index.html`.

[256] Special Level [online]. 2015. URL: `https://nethackwiki.com/mediawiki/index.php?title=Special_level&oldid=99258` [cited 2 Dec 2018].

[257] Friedrich Wilhelm Nietzsche. *Die Geburt der Tragödie [The Birth of Tragedy; or, Hellenism and Pessimism]*. George Allen & Unwin Ltd., London, Project Gutenberg edition, 2016. URL: `https://www.gutenberg.org/ebooks/51356`.

[258] Dierk Ohlerich. Nvscene 2008 session: From .kkrieger to debris. - procedural content generation. URL: https://youtu.be/QT2ftidLTn4.

[259] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2017. https://distill.pub/2017/feature-visualization. doi:10.23915/distill.00007.

[260] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 2018. https://distill.pub/2018/building-blocks. doi:10.23915/distill.00010.

[261] Martin O'Leary. The deserts of the west: A travel guide to unknown lands [online]. 2015. URL: https://github.com/dariusk/NaNoGenMo-2015/issues/156 [cited 27 Jan 2018].

[262] Martin O'Leary. Oisín: Wave function collapse for poetry, May 2017. URL: https://github.com/mewo2/oisin.

[263] Martin O'Leary. Twitter bio, 2017. URL: https://twitter.com/mewo2.

[264] Wesley Oliveira, Werner Gaisbauer, Michelle Tizuka, Esteban Clua, and Helmut Hlavacs. Virtual and real body experience comparison using mixed reality cycling environment. In Esteban Clua, Licinio Roque, Artur Lugmayr, and Pauliina Tuomi, editors, *Entertainment Computing – ICEC 2018*, pages 52–63, Cham, 2018. Springer International Publishing.

[265] Joseph Osborn, Adam Summerville, and Michael Mateas. Automatic mapping of NES games with Mappy. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 78:1–78:9, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3102071.3110576`, `doi:10.1145/3102071.3110576`.

[266] Derek Partridge and Yorick Wilks, editors. *Rational reconstruction as an AI methodology*, page 235–236. Cambridge University Press, 1990. `doi:10.1017/CBO9780511663116.022`.

[267] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[268] Ken Perlin. Real time responsive animation with personality. *IEEE transactions on visualization and Computer Graphics*, 1(1):5–15, 1995.

[269] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, 2002.

[270] Jon Peterson. *Playing at the World: A History of Simulating Wars, People and Fantastic Adventures, from Chess to Role-Playing Games.* Unreason Press, San Diego, CA, 2012.

[271] Christian Pinder. Elite: The New Kind, 1999-2002. URL: `http://www.new-kind.com/`.

[272] everest pipkin. A long history of generated poetics: cutups from dickinson to

melitzah, 2016. Archived by WebCite® at `http://www.webcitation.org/76fwxfAz5`. URL: `https://medium.com/@everestpipkin/a-long-history-of-generated-poetics-cutups-from-dickinson-to-melitzah-fce498083233`.

[273] Andrew Plotkin. Redwreath and goldstar have traveled to deathsgate: (or, the paarfi-o-matic loosed.) [online]. 2013. URL: `https://eblong.com/zarf/essays/r-and-g.html`.

[274] Gordon D Plotkin. A further note on inductive generalization. *Machine intelligence*, 6(101-124), 1971.

[275] Artem POPOV. Using perlin noise in sound synthesis. In *Linux Audio Conference 2018*, page 1, 2018.

[276] Stephen Prince and Wayne E. Hensley. The Kuleshov Effect: Recreating the classic experiment. *Cinema Journal*, 31(2):59–75, 1992. `doi:doi:10.2307/1225144`.

[277] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation.* TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL: `http://www.choco-solver.org`.

[278] PUBG Corporation. *PLAYERUNKNOWN'S BATTLEGROUNDS*, 2017.

[279] Miller Puckette. *The Theory and Technique of Electronic Music.* WORLD SCIENTIFIC, 2007. URL: `https://www.worldscientific.com/doi/abs/10.114`

2/6277, arXiv:https://www.worldscientific.com/doi/pdf/10.1142/6277, doi:10.1142/6277.

[280] Miller Puckette. *Pure Data*, 0.51-3, 2020. URL: https://puredata.info/.

[281] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021. URL: https://arxiv.org/abs/2103.00020, doi:10.48550/ARXIV.2103.00020.

[282] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. *openai.com*, 2018. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[283] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog [Internet]*, 2019. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

[284] Santa Ragione. *MirrorMoon EP*, 2013.

[285] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

[286] Herman Rapaport. *The Literary Theory Toolkit: A Compendium of Concepts and Methods.* Wiley-Blackwell, Chichester, West Sussex, United Kingdom, 2011.

[287] Aaron Reed. Aggressive Passive [online]. 2013. URL: `http://aaronareed.net/if/NaNoGenMo13/sample.html`.

[288] B. M. Reed. Poetics, western. In Roland Greene, Stephen Cushman, Clare Cavanagh, Jahan Ramazani, Paul F. Rouzer, Harris Feinsod, David Marno, Alexandra Slessarev, and Inc. ebrary, editors, *The Princeton Encyclopedia of Poetry and Poetics : Fourth Edition*, pages 1058–1064. Princeton University Press, 41 William Street, Princeton, New Jersey 08540, 2012. URL: `http://ebookcentral.proquest.com/lib/ucsc/detail.action?docID=913846`.

[289] Nora Reed. endless screaming [online]. 2015. URL: `https://twitter.com/infinite_scream`.

[290] Christoph Reinfandt. Reading texts after the linguistic turn: Approaches from literary studies and their implications. In Benjamin Ziemann and Miriam Dobson, editors, *Reading Primary Sources: The Interpretation of Texts from Modern History*, pages 37–54. Routledge, London, UK, 2009.

[291] Sebastian Risi and Julian Togelius. Procedural content generation: from automatically generating game levels to increasing generality in machine learning. *arXiv preprint arXiv:1911.13071*, 2019.

[292] Horst W. J. Rittel and Melvin M. Webber. Dilemmas in a general theory of

planning. *Policy Sciences*, 4(2):155–169, 1973. URL: `http://www.jstor.org/stable/4531523`.

[293] Timothy E. Roden and Ian Parberry. From artistry to automation: A structured methodology for procedural content creation. In *Entertainment Computing – ICEC 2004*, pages 151–156, 2004.

[294] T. J. Rose and A. G. Bakaoukas. Algorithms and approaches for procedural terrain generation - a brief review of current techniques. In *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, pages 1–2, 2016. `doi:10.1109/VS-GAMES.2016.7590336`.

[295] Mark Rosewater. When cards go bad, 1 2002. accessed 2020 Nov 4. URL: `http://magic.wizards.com/en/articles/archive/making-magic/when-cards-go-bad-2002-01-28`.

[296] Julian B. Rotter. Internal versus external control of reinforcement: A case history of a variable. *American Psychologist*, 45(4):489–493, 1990.

[297] John Ruskin. *The Nature of the Gothic: a Chapter of The Stones of Venice by John Ruskin.* Kelmscott Press, London and Orpington, 1892.

[298] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Chal-

lenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi:10.1007/s11263-015-0816-y.

[299] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 3 edition, 2016.

[300] Dominik Sacha, Michael Sedlmair, Leishi Zhang, John A. Lee, Jaakko Peltonen, Daniel Weiskopf, Stephen C. North, and Daniel A. Keim. What you see is what you can change: Human-centered machine learning by interactive visualization. *Neurocomputing*, 268:164 – 175, 2017. Advances in artificial neural networks, machine learning and computational intelligence. URL: http://www.sciencedirect.com/science/article/pii/S0925231217307609, doi:https://doi.org/10.1016/j.neucom.2017.01.105.

[301] Christoph Salge, Michael Cerny Green, Rodgrigo Canaan, and Julian Togelius. Generative design in minecraft (gdmc): Settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, pages 49:1–49:10, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3235765.3235814, doi:10.1145/3235765.3235814.

[302] Helena Sarin. *The Book of GANesis: Divine Comedy in Tangled Representations*. Helena Sarin, 12 2019. Limited run of 75 copies.

[303] Anurag Sarkar and Seth Cooper. Blending levels from different games using lstms.

In *The 5th Experimental AI in Games Workshop (EXAG)*, 2018. URL: `http://ceur-ws.org/Vol-2282/EXAG_125.pdf`.

[304] Rob Saunders and John S Gero. The digital clockwork muse: A computational model of aesthetic evolution. In *Proceedings of the AISB'01 Symposium on Artificial Intelligence and Creativity in Arts and Science*, volume 1, pages 12–21, University of York, Heslington, York, YOlO 5DD, England, 2001. Citeseer.

[305] Hugo Scurti and Clark Verbrugge. Generating paths with wfc. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[306] Carlo H Séquin, S Tager, B Vaysman, K Vetter, and Z Yang. *Procedural Modeling*. Computer Science Division (EECS), University of California, Berkeley, 1994.

[307] Noor Shaker, Mohammad Shaker, and Julian Togelius. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[308] Noor Shaker, Gillian Smith, and Georgios N. Yannakakis. *Evaluating content generators*, pages 215–224. Springer International Publishing, Cham, 2016. `doi:10.1007/978-3-319-42716-4_12`.

[309] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *NAACL-HLT (2)*, pages 464–468. Association for Computational Linguistics, 2018. URL: `https://aclanthology.info/papers/N18-2074/n18-2074`.

[310] Emily Short. The annals of the parrigues [online]. 12 2015. Self-published PDF. URL: https://emshort.blog/2015/12/07/procjam-entries-nanogenmo-and-my-generated-generation-guidebook/.

[311] Emily Short. *The Annals of the Parrigues*. Self-published PDF, December 2015. URL: https://inthewalls.itch.io/parrigues.

[312] Emily Short. Bowls of oatmeal and text generation [online]. September 2016. URL: https://emshort.blog/2016/09/21/bowls-of-oatmeal-and-text-generation/ [cited 12 December 2017].

[313] Emily Short. Emily Short – Five Strategies For Collaborating With A Machine [PROCJAM 2016] [online]. October 2016. URL: https://www.youtube.com/watch?v=narjui3em1k [cited 12 December 2017].

[314] SideFX. *Houdini*, 18.5. URL: https://www.sidefx.com/products/houdini/.

[315] SideFX. Houdini help 18.5 > nodes, 2020. URL: https://www.sidefx.com/docs/houdini/nodes/index.html.

[316] R. M. Smelik, T. Tutenel, K. J. De Kraker, and R. Bidarra. Semantic 3d media and content: A declarative approach to procedural modeling of virtual worlds. *Comput. Graph.*, 35(2):352–363, April 2011. URL: http://dx.doi.org/10.1016/j.cag.2010.11.011, doi:10.1016/j.cag.2010.11.011.

[317] Ruben Michaël Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra.

Interactive creation of virtual worlds using procedural sketching. In *Eurographics (Short papers)*, pages 29–32, 2010.

[318] Adam M. Smith. *Mechanizing Exploratory Game Design.* Ph.d. dissertation, University of California, Santa Cruz, December 2012. URL: `https://escholarship.org/uc/item/4600g227`.

[319] Adam M Smith. Strange loops in cfml: A livecoder's riddle. In *Proceedings of the International Conference on Computer Music (ICMC 2012)*, 2012.

[320] Adam M Smith, Eric Butler, and Zoran Popovic. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG*, pages 221–228, 2013.

[321] Adam M. Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, Sept 2011. `doi:10.1109/TCIAIG.2011.2158545`.

[322] Adam M. Smith and Michael Mateas. Computational caricatures: Probing the game design process with ai. In *Proceedings of the First International Workshop on Artificial Intelligence in the Game Design Process (IDP11)*, IDP11, 2011.

[323] Gillian Smith. Understanding procedural content generation: A design-centric analysis of the role of pcg in games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, page 917–926, New York, NY,

USA, 2014. Association for Computing Machinery. doi:10.1145/2556288.2557341.

[324] Gillian Smith. An analog history of procedural content generation. In *Proceedings of the 2015 Conference on the Foundations of Digital Games (FDG 2015)*, Monterey, CA, June 2015.

[325] Gillian Smith. Understanding the generated. In T.X. Short and T. Adams, editors, *Procedural Generation in Game Design*, chapter 22. Taylor & Francis, CRC Press, 2017. URL: https://books.google.com/books?id=_M8_MQAACAAJ.

[326] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 4:1–4:7, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1814256.1814260, doi:10.1145/1814256.1814260.

[327] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, pages 209–216, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1822348.1822376, doi:10.1145/1822348.1822376.

[328] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on*

Computational Intelligence and AI in Games, 3(3):201–215, Sept 2011. doi: 10.1109/TCIAIG.2011.2159716.

[329] Harvey Smith. The future of game design: Moving beyond deus ex and other dated paradigms, 2001. URL: https://www.witchboy.net/articles/the-future-of-game-design-moving-beyond-deus-ex-and-other-dated-paradigms/.

[330] Sam Snodgrass. *Markov Models for Procedural Content Generation*. PhD thesis, Drexel University, 2018.

[331] Sam Snodgrass and Santiago Ontañón. A hierarchical MdMC approach to 2d video game map generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, 2015.

[332] Sam Snodgrass and Santiago Ontañón. An approach to domain transfer in procedural content generation of two-dimensional videogame levels. In *Proceedings of AIIDE 2016*, 2016. URL: https://aaai.org/ocs/index.php/AIIDE/AIIDE16/paper/view/13985.

[333] Sam Snodgrass, Adam Summerville, and Santiago Ontañón. Studying the effects of training data on machine learning-based procedural content generation. In *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)*, pages 122–128, 2017.

[334] Joan Soler-Adillon. The open, the closed and the emergent: Theorizing emergence

for videogame studies. *Game Studies*, 19(2), 2019. URL: `http://gamestudies.org/1902/articles/soleradillon`.

[335] Tiago Boldt Sousa. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, volume 130, 2012.

[336] Francis Spufford. Masters of their universe [online]. October 2003. URL: `https://www.theguardian.com/books/2003/oct/18/features.weekend`.

[337] Francis Spufford. *The Backroom Boys: The Secret Return of the British Boffin*. Faber & Faber Ltd, 2004.

[338] Matt Stockham. Procedural Aesthetics—Building a toolset of aesthetic devices for generative games design. In *The Computer Games Journal 3*, number 2 in 3, pages 153–187. Springer, 2014.

[339] Oskar Stålberg. wave.html, May 2017. URL: `http://oskarstalberg.com/game/wave/wave.html`.

[340] Oskar Stålberg, Richard Meredith, and Martin Kvale. Bad North, 2018. Plausible Concept.

[341] Adam Summerville. Expanding expressive range: Evaluation methodologies for procedural content generation. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2018. URL: `https://aaai.org/ocs/index.php/AIIDE/AIIDE18/paper/view/18085`.

[342] Adam Summerville, Chris Martens, Ben Samuel, Joseph Osborn, Noah Wardrip-Fruin, and Michael Mateas. Gemini: Bidirectional generation and analysis of games via asp. In *Proceedings of the Fourteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE 2018. AAAI Press, November 2018.

[343] Adam Summerville and Michael Mateas. Super Mario as a string: Platformer level generation via LSTMs. *CoRR*, abs/1603.00930, 2016. URL: `http://arxiv.org/abs/1603.00930`, `arXiv:1603.00930`.

[344] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, Sep. 2018. `doi:10.1109/TG.2018.2846639`.

[345] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santi Ontañón Villar. The VGLC: the video game level corpus. *CoRR*, abs/1606.07487, 2016. URL: `http://arxiv.org/abs/1606.07487`, `arXiv:1606.07487`.

[346] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In Noah Wardrip-Fruin and Nick Montfort, editors, *The New Media Reader*, page 111–126. MIT Press, 2003.

[347] William Robert Sutherland. *The on-line graphical specification of computer procedures.* PhD thesis, Massachusetts Institute of Technology, 1966.

[348] Steve Swink. *Game feel : a game designer's guide to virtual sensation.* Morgan Kaufmann Publishers/Elsevier, Amsterdam ;, 2009.

[349] Brian Teasley, Daniel Farris, and The Octopus Project. The living venue audio project, 2015. URL: `http://www.saturnbirmingham.com/livingvenueproject/`.

[350] The Foundary. *Nuke*, 12.0v1. URL: `https://www.foundry.com/products/nuke`.

[351] Julien @Orteil42 Thiennot. thanks to procedural generation, i can produce twice the content in double the time, 11 2016. URL: `https://twitter.com/orteil42/status/802258188498333701`.

[352] Tommy Thompson. The director ai of left 4 dead | ai and games. URL: `https://youtu.be/WbHMxo11HcU`.

[353] Julian Togelius, Alex J Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O Stanley. Procedural content generation: Goals, challenges and actionable steps. In *Artificial and Computational Intelligence in Games*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[354] Julian Togelius and Noor Shaker. The search-based approach. In *Procedural Content Generation in Games*, pages 17–30. Springer International Publishing, 2016. `doi:10.1007/978-3-319-42716-4_2`.

[355] Julian Togelius, Noor Shaker, and Mark J. Nelson. Introduction. In *Procedural*

Content Generation in Games, pages 1–15. Springer International Publishing, 2016. doi:10.1007/978-3-319-42716-4_1.

[356] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[357] Michael Toy, Glenn Wichman, Ken Arnold, and Jon Lan. *Rogue*, 1980.

[358] Mike Treanor, Bobby Schweizer, Ian Bogost, and Michael Mateas. Proceduralist readings: How to find meaning in games with graphical logics. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, FDG '11, pages 115–122, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/2159365.2159381, doi:10.1145/2159365.2159381.

[359] Anna Lowenhaupt Tsing. *The mushroom at the end of the world: On the possibility of life in capitalist ruins*. Princeton University Press, 2015.

[360] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230, arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230, doi:https://doi.org/10.1112/plms/s2-42.1.230.

[361] Aaron van den Oord, Oriol Vinyals, and koray kavukcuoglu. Neural discrete representation learning. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: `https://proceedings.neurips.cc/paper/2017/file/7a98af17e63a0ac09ce2e96d03992fbc-Paper.pdf`.

[362] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008. URL: `https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf`.

[363] Thomas F. Varley and Erik Hoel. Emergence as the conversion of information: a unifying theory. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 380(2227):20210150, 2022. URL: `https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2021.0150`, `arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2021.0150`, `doi:10.1098/rsta.2021.0150`.

[364] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017. URL: `https://arxiv.org/abs/1706.03762`, `doi:10.48550/ARXIV.1706.03762`.

[365] Dan Ventura. Mere generation: Essential barometer or dated concept. In *Proceed-*

ings of the Seventh International Conference on Computational Creativity, pages 17–24, 2016.

[366] Vitruvius. *Vitruvius: The Ten Books on Architecture.* Cambridge: Harvard University Press, 1914.

[367] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam M. Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2018)*, New York, NY, USA, July 2018. ACM. URL: `http://doi.acm.org/10.1145/3205455.3205517`, `doi:10.1145/3205455.3205517`.

[368] John Von Neumann. Various techniques used in connection with random digits. In *John von Neumann, Collected Works*, volume 5, pages 768–770. Pergamon Press, 1963.

[369] John Von Neumann. The theory of automata: Construction, reproduction, homogeneity. In Arthur W. Burks, editor, *Theory of Self-Reproducing Automata.* University of Illinois Press, Urbana and London, 1966.

[370] Noah Wardrip-Fruin. *Expressive Processing: Digital Fictions, Computer Games, and Software Studies.* The MIT Press, 2009.

[371] Georgia Warnke. Hermeneutics, Nov 2016. Published Online. Accessed 2019 April 17. URL: `http://oxfordre.com/literature/view/10.1093/acrefore/978019`

0201098.001.0001/acrefore-9780190201098-e-114, doi:10.1093/acrefore/9780190201098.013.114.

[372] Aline Weber, Lucas N Alegre, Jim Tørresen, and Bruno Castro da Silva. Parameterized melody generation with autoencoders and temporally-consistent noise. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 174–179. Universidade Federal do Rio Grande do Sul, 2019. URL: http://urn.nb.no/URN:NBN:no-80503.

[373] Brent Werness. *Bot Ross*, commit 93afa41832c4a29c80f17394b177f79f414fe0e0, 2017. URL: https://bitbucket.org/BWerness/bot-ross.

[374] Mick West. A shattered reality. *Game Developer*, pages 34–36, 8 2006. URL: http://twvideo01.ubm-us.net/o1/vault/GD_Mag_Archives/GDM_August_2006.pdf.

[375] Glenn R. Wichman. A brief history of "Rogue" [online]. 1997. URL: https://web.archive.org/web/20160315120630/http://www.wichman.org/roguehistory.html.

[376] Anna Wiener. The weird, analog delights of foley sound effects, June 2022. URL: https://www.newyorker.com/magazine/2022/07/04/the-weird-analog-delights-of-foley-sound-effects.

[377] Giles Williams and the Oolite Project. Oolite, 2003–2022. URL: http://www.oolite.org/.

[378] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Inc., 2002. URL: `https://www.wolframscience.com/nks`.

[379] Andrea Wollensak, Judith Goldman, and Bridget Baird. Ice core modulations: Performative digital poetics. In *Proceedings of the XVIII Generative Art Conference*, GA2015, 2015. URL: `http://www.generativeart.com/ga2015_WEB/ice-modulation_Wollensak_Baird.pdf`.

[380] Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 291–294, New York, NY, USA, 1996. Association for Computing Machinery. `doi:10.1145/237170.237267`.

[381] Will Wright. Dynamics for designers. In *Game Developer's Conference 2003*, 2005. URL: `https://www.gdcvault.com/play/1019938/Dynamics-for`.

[382] Will Wright. The future of content. In *Game Developer's Conference 2005*, 2005. URL: `https://www.gdcvault.com/play/1019981/The-Future-of-Content-%2528English`.

[383] Shaoyou Xie, Wei Zhou, and Honglei Han. Antagonistic procedural content generation of sparse reward game. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021*, pages 1–4, 2021.

[384] Georgios N Yannakakis, Antonios Liapis, and Constantine Alexopoulos. Mixed-initiative co-creativity. In *Proceedings of the 9th International Conference on the*

Foundations of Digital Games. Society for the Advancement of the Science of Digital Games, 2014.

[385] Derek Yu. *Spelunky*. Boss Fight Books, Los Angeles, CA, 2016.

[386] Adeel Zafar, Hasan Mujtaba, and Mirza Omer Beg. Search-based procedural content generation for gvg-lg. *Applied Soft Computing*, 86:105909, 2020.