40

Procedural Content Generation

An Overview

Gillian Smith

40.1 Introduction

Procedural content generation (PCG) is the process of using an AI system to author aspects of a game that a human designer would typically be responsible for creating, from textures and natural effects to levels and quests, and even to the game rules themselves. Therefore, the creator of a PCG system is responsible for capturing some aspect of a designer's expertise—a challenging task for an AI!

PCG has been used in many games for several different purposes. Two popular motivations are replayability and adaptability. PCG can provide large amounts of content, so that each time the player starts the game, they will have a different experience. In combination with an AI system that can infer player skill, PCG can be used as a form of dynamic difficulty adjustment, shifting the content the player will see in order to adapt to their skill level.

One of the first examples of PCG was in the game *Elite* [Braben 84], where entire galaxies were generated by the computer so that there could be an expansive universe for players to explore without running afoul of memory requirements. However, unlike most modern games that incorporate PCG, *Elite*'s content generation was entirely deterministic,

allowing the designers to have complete control over the resulting experience. In other words, *Elite* is really a game where PCG is used as a form of data compression. This tradition is continued in *demoscenes*, such as *.kkrieger* [.theprodukkt 04], which have the goal of maximizing the complexity of interactive scenes with a minimal code footprint. However, this is no longer the major goal for PCG systems.

Regardless of whether creation is deterministic, one of the main tensions when creating a game with PCG is retaining some amount of control over the final product. It can be tempting to use PCG in a game because of a desire to reduce the authoring burden or make up for missing expertise—for example, a small indie team wanting to make a game with a massive world may choose to use PCG to avoid needing to painstakingly hand-author that world. But while it is relatively simple to create a system that can generate highly varied content, the challenge comes in ensuring the content's quality and ability to meet the needs of the game.

There are several major approaches to PCG that offer different means for this control, as well as supporting different extents to which the control can take place. There are also many reasons for why PCG may be used in a game, which can help inform a game designer's and/or developer's choice about what technique to use when creating the system. *Civilization* [MicroProse 91], *Rogue* [Toy 80], *Minecraft* [Persson 11], and *Borderlands* [Gearbox Software 09] are all examples of popular games that use PCG, but with completely different approaches and purposes.

For the sake of scope, this chapter focuses on creating content that a player somehow interacts with as part of gameplay—think levels and quests, rather than textures and trees that involve procedural modeling [Ebert 03]. It will also avoid a discussion of procedural generation of game rules—that is, research that is really still in its infancy as of this book's publication [Smith, A. 10, Togelius 08]. The focus is on the procedural creation of content (e.g., what would be created by a level designer) rather than on storytelling and NPC behavior, though there are many relationships between these systems. However, while we won't delve into systems that create game rules, for many PCG systems, it is still necessary to find ways to formally specify aspects of the game's rules in order to guarantee that playable content will be generated.

This chapter will give an overview of PCG, survey different techniques for creating content generators, examine how PCG fits into a game's design, and give some advice for how to choose an appropriate method for a game. At the end, there are pointers to literature and resources where the reader can find more details on PCG and learn how to stay up to date on PCG research.

40.2 Technical Approaches to Content Generation

While there are no off-the-shelf tools or frameworks for creating your own PCG system, there are several common approaches and methods for knowledge representation. This section will give an overview of these approaches and methods and discuss some tradeoffs between them.

40.2.1 Algorithms and Approaches

One of the primary considerations when choosing an approach to designing a content generator is the extent and kind of control needed. Approaches to PCG range from purely

bottom-up, simulation-based methods that only permit control over the initial state of the world and the operators for changing it to top-down, constraint-driven methods that let the system meet firm authoring constraints but may be more expensive to create and debug.

40.2.1.1 Simulation Based

Simulation-based approaches to PCG start with an initial world and a set of operators that can alter that world and then run a simulation for a predetermined period of time. For example, a simulation-based approach to creating terrain might start with a single landmass in an ocean, a climate/rainfall model, and an erosion model to create a new terrain with rivers, lakes, cliffs, and beaches. *Dwarf Fortress* [Bay 12 Games 06, Adams 15] is a game that takes this approach to generation, with a full simulation that creates landmasses, caves, and a history for the world. Simulation-based approaches do not permit any control over the final output of the system without using a generate-and-test paradigm (described further in the section on constructionist systems). While simulation can be a slow process with limited control, it has two potential benefits that may make it a good choice: (1) it provides you with a content history that can be referred to or replayed, and (2) it can be run during gameplay to create a world that reacts to the player's choices and actions.

40.2.1.2 Constructionist

A constructionist approach is one that pieces together premade building blocks according to an algorithm especially constructed for the game. This algorithm has design knowledge implicitly baked into it—any intelligence in the process exists only in the choices made by this one algorithm, rather than more explicitly stated as an optimization function or as a set of design constraints. For example, many *Rogue*-like level generators [Rogue Basin 12] use constructionist approaches that build different rooms and then construct corridors between them. Constructionist approaches are typically ad hoc, specialized for one particular game with little applicability beyond it, and accessible primarily to developers, rather than designers and artists.

Constructionist approaches often rely entirely on their knowledge representation (see below). Many, though not all, approaches involve taking large, preauthored pieces of content and placing them next to each other randomly, as in endless runner games like *Robot Unicorn Attack* [[adult swim] games 10] or *Canabalt* [Saltsman 09]. This kind of constructionist approach is perhaps more accurately named *content selection*, where there is no attempt to make intelligent choices in the *process* that is followed, but there is very tight designer control over the building blocks.

Constructionist systems are difficult to control and difficult to get good variation from; it's easy for all the content to start to feel the same if there's a limited set of building blocks to choose from. Large branching control flow structures can also be difficult to debug, especially if the bug lies in a piece of code that is executed very rarely. And you can spend a huge amount of time tweaking your algorithm to get content that is *just right* in one situation, without realizing that you're simultaneously breaking the content for some different situation.

Designing your own algorithm from the ground up does have some benefits, however. From a design perspective, content selection is a lightweight approach that is good enough for some games, especially where there aren't a lot of constraints on what constitutes

playability (e.g., a simple platforming game such as *Canabalt* just needs the platforms to be reachable, whereas a lock-and-key puzzle game such as Joris Dormans's *Zelda*-like game [Dormans 10] has deeper constraints).

40.2.1.3 Grammars

A grammar-based approach starts by specifying the possibility space of content (or *generative space*) as a formal grammar. Next, an interpreter for that grammar (which will parse and process the rules to create content) is built. This explicit separation of grammar rules from content assembly offers more organization than a constructionist approach, where the rules are often implicitly defined in the code, and can also be useful to technical designers who may be able to edit the grammar rules without needing to touch the content assembly system. Grammars have been used to create levels for platforming games [Smith, G. 11c] and action-adventure games [Dormans 10], as well as in tools for rapidly designing buildings [Müller 06]. Shape grammars are also a useful kind of grammar for creating visual content [Stiny 80].

When used in a process in which content is tested against requirements and discarded if it doesn't pass the tests, this approach strikes a balance between bottom-up emergence from designer-specified rules (in the form of the grammar) and top-down control over content (in the form of the acceptance rules for the generated content). It still allows a computer to explore the design space and come up with surprising and varied results.

Authoring the rules for a grammar-based system is done by breaking down your desired content into increasingly small pieces. The grammar will then specify rules as nonterminal and terminal symbols. Nonterminal symbols can be expanded into more nonterminals and terminals, specified by a set of rules. Content creation begins with a single nonterminal symbol, the start symbol, and then repeatedly replaces nonterminals with new symbols (which may be terminals, nonterminals, or a mix of the two) according to a set of expansion rules until only terminal symbols remain. There may also be more than one rule per nonterminal symbol, thus allowing nondeterministic content to be created. Some rules might be weighted toward being selected more frequently than others, which can provide some additional control over the kind of content that will be produced.

The grammar's interpreter is responsible for selecting the next rule to expand. It can be tempting to combine the grammar rules and the interpreter into one big branching control flow, and for simple grammars this can suffice. However, the power of grammars often lies in the ability to quickly add or change rules and see the impact on content, and grammars often change and grow to meet changing design constraints. Thus, it is a good idea to keep the interpreter and the grammar separate.

A grammar can produce huge amounts of content often very quickly. Unfortunately, grammars are prone to *overgeneration*, where they may create content that was not intended when the grammar rules were designed. If the grammar is too loosely constrained, there is no guarantee that everything it can create will be *good* content. Fixing overgeneration in the grammar rules can sometimes restrict the abilities of the grammar too much, leading to *undergeneration*—all the content may now be considered good, but it will also feel too similar and predictable. Thus, choosing to overgenerate and then running a suite of tests on the results can ensure that surprising content is still generated, but unacceptable content is discarded.

In order to achieve this, there must be defined some simple-to-measure acceptance criteria (e.g., that a level must have at least 15 features) with which the bad content will be culled. These criteria also offer an opportunity to explicitly define criteria that are difficult, or even impossible, to specify in the grammar rules, and provide some ability to have top-down design constraints while still benefiting from the emergent content that comes from using bottom-up rules for generation.

The strength of a grammar-based approach lies in the combination of simple, easy-to-author rules that provide emergent and interesting content with a generate-and-test paradigm that lets the generator meet soft constraints. Grammar rules are also easy to use to express design patterns. Since grammars usually generate content very quickly, generating a great deal of content and throwing much of it away are not expensive. As an example, *Launchpad* [Smith, G. 11c] is an unoptimized level generator for Mario-like platforming levels. Its grammar-based representation is capable of creating ten thousand candidate levels in only a few seconds. The speed of this approach is dependent in large part upon the complexity of the grammar rules, but since it is possible to specify recursive rules, speed is also dependent upon the allowed recursion depth. The weakness of grammars is their difficulty in meeting hard design constraints and in debugging the rule system, which can become quite complex.

40.2.1.4 Optimization

An optimization-based generator involves a search process (often unbounded) that seeks out the optimal combination of components according to some evaluation function. This evaluation function is usually specified as a formula that attempts to calculate the desirability, and the search attempts to maximize this value. Alternately, there can be a *human in the loop,* with a human player or designer selecting their favorite content from among the candidates.

Evolutionary algorithms are a popular approach to optimization-based content generation in academic research. They attempt to mimic natural evolution. An initial population is created and then bred and mutated into a new population that can be evaluated by the fitness function (or by the human in the loop). The best candidates are bred and mutated once again, and their children are evaluated. This cycle continues until there is a piece of content that is a good fit to the evaluation function or until a maximum number of candidate generations have been created, at which point the candidate with the best evaluation score is chosen.

As a simple example of an evolutionary algorithm, consider the problem of procedurally generating a maze. An evolutionary algorithm might start with a population of a thousand completely random mazes. It would then calculate the fitness score for each maze based on how well that particular candidate meets the evaluation criteria and would generate a new population of mazes by *breeding* highly rated mazes together—perhaps by combining the left and right halves of two different mazes into a new one. This process would be repeated either until a predetermined amount of time has passed or until a maze that meets the acceptance criteria has been generated.

There are a lot of nuances to creating genetic algorithms. Optimization-based approaches are also highly sensitive to the knowledge representation used (e.g., representing a maze as a set of walls with endpoints and lengths or as a grid with open and closed cells) [McGuinness 12], as well as to the particular implementation of the algorithm.

What should the mutation and breeding operators be? How should the evaluation function be crafted? How should the percentage of the population that is chosen to breed versus mutate be selected? Should some of the candidates be saved for the next generation without breeding (known as elitism)? There is a survey article on search-based methods for PCG that provides a good entry point to the literature [Togelius 11].

All optimization-based approaches to PCG must have an evaluation function—some way of quantifying the overall goodness of an arbitrary piece of content. These evaluation functions can be fairly simple approximations, such as the distance from the actual distribution of components in the content to the desired distribution of those components. However, it can be difficult to capture everything important about the content in one evaluation function. *Player experience modeling* [Yannakakis 11] is an attempt to treat the evaluation function more abstractly by learning a model of individual player preferences by having them play some of the content and then applying the model as the fitness function to create personalized content. A simpler use of player modeling is to use the human as the evaluation function. *Galactic Arms Race* is a game that takes a *human-in-the-loop* approach and uses an inferred simple model of player preference (those weapons the player uses most often are most desirable) to generate new weapons that are personalized to a particular play style [Hastings 09].

The use of an optimization-based approach relies on a comfort with the notion that there is such a thing as an optimal piece or pieces of content and that it is possible to express this mathematically. Note that this evaluation function does not necessarily need to be reduced to a mathematical definition of *fun*. It can instead be a set of desired properties that the system should aim for. If the generator is being designed with a concrete goal that can be expressed mathematically, then this approach may be a good one. It can also work quite well if there is a way to involve a human in guiding the generation process at runtime. Another benefit is that, like the generate-and-test approach used with grammars, optimization-based approaches can be useful for soft constraints—properties of content that are desirable but not crucial to the function of the game. However, evolutionary algorithms that do not have a human-in-the-loop can be slow, and player experience modeling requires the player to be put through training levels, which can be time consuming for the player and must be designed into the game.

40.2.1.5 Constraint Driven

Constraint-driven methods are declarative approaches in which hard design constraints are specified, and then a constraint solver is used to find all potential solutions that meet those constraints. All of the content is expressed as variables with ranges of potential values, with constraints dictating the relationships between these variables. This entirely top-down approach allows the specification of knowledge about what the content *should* look like separately from the underlying search algorithm.

Constraint satisfaction has been used for generating room interiors [Tutenel 09] using semantic constraints, which introduce knowledge about what objects are and how they relate to others (e.g., a table should be surrounded by chairs). Numerical constraint solving has been used for placement of platforms and other level geometry in platformer levels [Smith, G. 11b]. Answer set programming, a method for specifying constraint problems in first-order logic, has been used for levels in an educational puzzle game [Smith, A. 12] as well as the real-time strategy game *Warzone 2100* [Smith, A. 11].

The challenge in constraint satisfaction for content generation comes from fully specifying all of the constraints. Commonsense constraints, such as the idea that two objects cannot occupy the same position at the same time, must be specified along with more game-specific constraints. However, constraints are a powerful method for expressing a design space, and this approach works well when there are many constraints that *must* be met in order for the content to be acceptable. Runtime performance for constraint satisfaction approaches to PCG varies drastically based on the size of the problem, how the problem is described, and the number of constraints. Many solvers, such as answer set programming, work by reducing the problem to a Boolean satisfiability problem, which is an NP-complete problem. However, for small-to-medium-sized domains, such as mazes, grid-based puzzle games [Smith, A. 11], and even simple platformers [Smith, G. 11b], constraint-based content generation can produce results in a matter of seconds, or a few minutes at worst. What's more, adding constraints can actually improve the runtime performance of some constraint-based systems, as it allows the system to rapidly eliminate portions of the search space that don't contain satisfactory solutions.

40.2.2 Knowledge Representation

Many of the approaches described in the previous section can be further varied by changing the knowledge representation—that is, the building blocks that the content generator will piece together. Here, there are four major kinds of building blocks that trade off authoring control against risk of the player recognizing common patterns. They are presented in order from most to least human authoring.

40.2.2.1 Experiential Chunks

An experiential chunk captures a sufficiently large amount of content that, on its own and outside the context of the entire piece of content, it could still be experienced by the player as its own entity. An example would be the level chunks used in *Robot Unicorn Attack*. One advantage of this representation is that there is a great deal of artistic and design control over the appearance of the generated content, but there is a significant chance that the player will begin to notice the same chunks repeated again and again. Raph Koster notes that game players are pattern recognition machines [Koster 04], and this is certainly true for PCG. Unless pattern recognition is desired, experiential chunks should be either avoided or tempered by using a mixture of approaches to designing the generator (see Section 40.2.3). Experiential chunks are usually used with constructionist algorithms, but can also be used with grammars.

40.2.2.2 Templates

Templates are a more generalized form of experiential chunk, where the design team can still control the content, but leaves blanks for the AI to fill in automatically. Templates are like the *Mad Libs* of PCG and, unless care is taken to construct the template and the rules for what content can fill in the gaps, can have the same quirky consequences as the word game. However, templates can strike a nice balance between authorial control and variety for high-fidelity games.

Templates are a kind of high-level design pattern, and the design pattern literature can be a good place to draw inspiration for templates for a generator. Bjork and Holopainen's book [Bjork 04] is a good collection of general patterns, but there are also patterns specific

to genres such as first-person shooters [Hullett 10], role-playing game levels and quests [Smith, G. 11a, Onuczko 05], and 2D platformers [Smith, G. 11c], to name a few examples.

40.2.2.3 Components

Like templates, components are patterns that have been designed by a human. However, unlike templates and experiential chunks, components cannot stand on their own to be experienced as content in their own right. For example, enemies in a first-person shooter have general behavior dictated by a human designer, but cannot exist without the broader context of the level that they inhabit. Using components reduces the risk that content patterns will be seen by the player. However, it also means that the generation algorithm needs to play a much stronger role, as it will be almost entirely responsible for the quality of the content that is produced. Component patterns can work well with all of the generation approaches mentioned in this chapter.

40.2.2.4 Subcomponents

A subcomponent representation uses the smallest possible assets. Subcomponents do not have embedded semantic information about what the content is or how it should be used. They can be imagined as the same kind of building block that humans use to construct their levels, such as art assets from a tileset. Very few generators use this approach to content generation because it is difficult for the generator to understand how to piece things together without some amount of semantic information about the content, and thus it's more common for generators to use a component representation. For example, it is helpful for a generator to understand the concept of a *room* or a *chest* and how they are allowed to fit together. However, *Galactic Arms Race* [Hastings 09], an experimental game that creates particle system weapons, is one example of a game that uses this representation.

40.2.3 Mixing and Matching

These approaches and knowledge representation techniques can be combined to produce more sophisticated systems and to meet the demands of the game's design. For example, using content selection to create levels at runtime that use pregenerated pieces of content (i.e., experiential chunks) provides the reduced authoring burden and high variety of PCG while still allowing for a lightweight generation algorithm; furthermore, the use of pregenerated content pieces can provide more varieties among the chunks than would be easy for human designers to create. *Polymorph* [Jennings-Teats 10] is an experimental adaptive level generator that uses this approach to create levels that are customized to a player's abilities. Each pregenerated chunk is tagged with a difficulty score, and as the player progresses through the level, chunks with an appropriate difficulty score (based on an estimate of player skill level via observing their failures) are placed in front of the player.

Using different content generation techniques at different layers of abstraction can help balance out human authoring with algorithm complexity. For example, a constructionist, template-based approach to create a dungeon crawler level could be combined with a constraint solver to place items into the rooms in the slots that are left unfilled. This would allow for tight control over the room's overall appearance and ensure that gameplay requirements are met, but still provide a high level of variety in the room's contents. This hypothetical generator is similar to the generation approach taken in the

Tanagra level design assistant [Smith, G. 11b], where reactive planning is used in conjunction with a numerical constraint solver to create platformer levels in collaboration with a human designer.

Mixing-and-matching paradigms can be very powerful, but it does have one major drawback: in general, different layers of the generator will not be able to easily communicate with each other. In our dungeon crawler example given, if the constraint solver is unable to place items into the room to meet all design constraints, the generator must go back and choose another set of room templates, then ask the constraint solver to try again. This is because the constraint solver does not have the power to alter room layout, and the room layout algorithm does not have any knowledge of the kind of constraints that the solver cares about. In a purely constraint-based system, the room layout would potentially be able to shift to meet constraints on the level components.

40.3 Understanding PCG's Relationship to a Game

Now that we have an understanding of the different ways to build a PCG system, let's look at how to understand the role that PCG will be taking in the game. This is largely a set of design decisions, but it has a lot of impact on the way that the AI system should be constructed.

40.3.1 PCG's Mechanical Role

There are several important aspects of PCG's mechanical role within the game, including how it is integrated in to the overall game, if and how the player will interact with it, and how well it can be controlled by designers and players.

40.3.1.1 Game Stage

Does content generation need to happen *online*, while the player is playing? Or can it occur *offline*, while the level is loading or even while the game is being developed? If the generator must run online and frequently, then performance is one of the largest concerns, and it may be necessary to compromise quality to some extent. On the other hand, a generator that runs offline can be slower, but it needs to be able to create content that is flexible enough to support the variety that is inherent in player behavior (i.e., content can't be crafted as a response to the player's actions), and it also has to store and load generated content efficiently.

40.3.1.2 Interaction with the Generator

In games where the player does interact with the generator (as opposed to just with the generated content), there are three main types of interaction the player may have: *parameterized*, *preference*, and *direct manipulation*.

Parameterized control lets the player set some parameters that influence the generator before they interact with content. For example, the *Civilization V* level generator allows the player to set parameters such as world age, temperature, and landmass type.

Preference control means that the player can (directly or indirectly) specify preferences for what content they will see next while they are in the game. For example, *Galactic Arms Race* infers player preferences based on their behavior, allowing them to provide indirect control over the kind of weapons that they will see next.

Direct manipulation lets the player directly interact with the content while the generator runs in the background to assist the player. This kind of control appears in *Spore's* creature creator, where the player builds a model of a creature while the content generator supports the player's choices and augments them by supplying textures and animations.

40.3.1.3 Control over Player Experience

While it is possible to build a content generator that simply stitches together preauthored chunks of content (i.e., a constructionist approach), it is often desirable to have tight design control over particular aspects of the player's experience. This control can come in two forms: *compositional* and *experiential*.

Compositional control means that the generator can make design guarantees about the presence of particular components in the final product, for example, a platformer level generator that can guarantee that 50% of the challenges in a level will be due to gaps or a quest generator that can guarantee the quest will involve the player finding two angry shopkeepers.

Experiential control means that the generator has an understanding of some aspect of the player's in-game experience beyond just component placement, for example, a platformer generator that can guarantee level pacing independently from what components are present or a quest generator that can guarantee that a quest will have a particular difficulty associated with it.

40.3.2 Player Interaction with PCG

The *dynamics* of PCG (borrowed from Hunicke et al.'s mechanics, dynamics, and aesthetics framework for analyzing games [Hunicke 04]) are the styles of play that arise from the player's interactions with the generated content. Understanding these patterns can help in selecting which techniques to use and the range of content to create.

40.3.2.1 PCG Relationship to Other Mechanics

We can first consider the role of PCG in the game relative to the other game mechanics. Will the player's experience revolve around the generated content, or are we creating decorative, incidental content that only augments the player's experience? Some games use PCG to frame player experience: *Civilization V* is a game where procedurally generated maps give the player something surprising to explore in the early phase of the game, but most of the player's strategies revolve around other aspects of the game (e.g., build order and military strategy)—and indeed, many scenarios with fixed, designer-generated maps exist. Endless runner games such as *Canabalt*, on the other hand, have generators that the player's experience is entirely dependent upon. These games have no mechanic other than to interact with the generated content. Understanding the extent to which your generator will influence the player's experience can help you decide how to focus your efforts, how important design guarantees are, and how to ensure that there is sufficient variety for players.

40.3.2.2 Reacting

PCG is often used to create surprising or unexpected content, forcing the player to react to unforeseen circumstances rather than allowing them to regurgitate memorized actions. This reaction may be related to exploration, as in *Spelunky* [Yu 09], or it may be a test of

reaction time, as in *Robot Unicorn Attack*. Reaction comes from the random elements of PCG, but you may wish to be able to control this somewhat. If the game would benefit from the player being able to practice some aspects, but still be a test of reaction in others, then using experiential chunks may be a good choice for your knowledge representation.

40.3.2.3 Strategizing

The use of a content generator that runs online and can be controlled (either directly or indirectly) by the player leads to situations where the player forms strategies about how to interact with the generator. If this style of play is desirable, then consider what these strategies should be when designing the generator. For example, if the player should be able to strategize around the composition of challenges in levels, as in *Endless Web* [Smith, G. 12], then the generator must be designed such that this can be explicitly controlled.

40.3.2.4 Searching

Players can do two kinds of searching as a result of PCG. The first is when the content being generated is a world: players will search through it and explore it, looking for surprises and interesting moments, as in *Minecraft*. The second is when the content is smaller and can be found in the environment, when players may search for interesting or unique content, as in *Borderlands*. The lesson to be learned from this dynamic is the same no matter the kind of searching: the generator should produce content that can be surprising and exciting for the player to find. A generator may be able to produce millions of unique pieces of content, but the quantity is meaningless if those million pieces all *feel* the same to the player. Introducing surprise and variety can be difficult. It may come from incorporating small, hand-authored pieces of content into the generator for the player to find or from careful construction of grammar rules and weights.

40.3.2.5 Practicing

Many games use PCG to allow the player to practice game strategies. *Civilization V* (and other strategy games with generated maps) is a particularly good example of this dynamic—the other mechanics of the game provide multiple strategies for success, and the procedural placement of land and resources means that the player can practice selecting from among those strategies and adapting them to many different, but controlled, environments. In such games, especially multiplayer games, the ability for the generator to make design guarantees about resource availability, playability, and balance may be of high importance. Constraint-based generators, or generators that use generate-and-test to approach design requirements, are good techniques for games that need such guarantees.

40.3.2.6 Community

Finally, PCG can be used to vary the player's experience so as to make it impossible to write a walkthrough for your game. This leads to players having different experiences from each other. When crafting a PCG system, it is common to think of a single player's experience with the generated content, but for some games, it is also important to think about the *community*'s experience with that content. Fans of games such as *Dwarf Fortress* engage in long, drawn-out discussions and debates based on the results the generator provides, telling stories of their own experiences in their own unique world.

40.4 Choosing an Approach

This section discusses how to get started with PCG and how to choose an approach, as well as the kinds of design considerations that need to be taken into account.

40.4.1 Getting Started

There are two ways to start building a PCG system: from the bottom up or from the top down. Doing both of these at the same time may be helpful, as one method may inspire decisions you make in the other.

A top-down approach starts by mapping out the kinds of content you want to be able to create and developing specific examples of what the generator should be capable of. It can be particularly helpful to start by designing examples of content that represents the extremes of what the generator could produce: *What are the easiest and hardest levels, or the quirkiest weapons, or the ugliest and prettiest flowers?* If there are parameters that the player should have control over (e.g., the size of the map or the shape of the robot), these are useful starting points for building examples of what the generated content should look like. From these examples, start distilling patterns: these will be the building blocks in the generator.

A bottom-up approach works up from the tools. For example, if you think a grammar approach is an interesting choice, start writing some production rules and see what kind of content you can get out of it. Keep adding rules, and tweaking the ones that you already have, until it starts to feel like you are getting what you want. If a constraint system interests you, start writing simple playability constraints, look at how they can go wrong, and then iterate on them, adding more or changing the ones you have until you get the sort of content that you want to see.

For both approaches, remember that the algorithms you use and building blocks you are piecing together define a large probability space, where each point in that space is an individual piece of content. The space may have some strange twists and turns—there may be kinds of content that you can never produce without changing your knowledge representation or your algorithm, and there may be kinds that you overproduce until you rein in the generator. Rather than thinking about how to create a single perfect piece of content, think about the space of potential content that can come out of your generator.

40.4.2 Game Design Constraints

One of the difficulties that comes with PCG is the need to relinquish some amount of authorial control. The ability for the human designer to control the content that the player will see is limited by the need to allow it to be generated dynamically. With this in mind, when selecting PCG techniques, you should consider (1) what kind of design control is needed and (2) if there is a need for tight design guarantees or if it is sufficient to come close. For example, a level generator probably has a need to absolutely guarantee that levels are playable, but it may be sufficient for the levels to just come close to meeting difficulty or pacing requirements. For that matter, even the playability guarantee might not be required if the player is given tools to morph the environment or if the playability issues occur infrequently. These constraints may be dealt with in a different way depending on how important they are. For example, the *Launchpad* level generator [Smith, G. 11c] guarantees level playability by baking it in to the generation algorithm, allowing only legal

placement of level geometry, but it does not guarantee that the desired frequency of different level components will be perfectly met due to its use of generate-and-test for that design constraint.

Other potential game design constraints arise when there is a need to obey rules imposed by some other aspects of the design. For instance, if there is an overarching game narrative that all generated levels must fit into, then there must be a way to express the important aspects of narrative as a constraint or rule for the generator.

40.4.3 Relationship with Art

Assuming that some or all of the content that a PCG system produces is visual in nature, the PCG system must be designed in concert with artists. Unless all of the content, including textures, will be generated by the computer (a difficult challenge), artists must create modular assets for the generator to use.

Art is one of the main motivations behind the different kinds of building blocks being chosen: experiential chunks are good when the art direction mandates a rich and detailed world and the artists want tight control over its appearance. Each chunk needs to be designed so that it can fit with others, but otherwise they can stand alone. At the other extreme, a subcomponent representation is useful for a game that uses a tile-based representation, such as a 2D sidescroller, so each subcomponent has a single tile art asset that can be associated with it.

It may still be important for the generator to incorporate some sense of art direction, so that the structure of generated content is suitable to be skinned by human-created art assets. For instance, in a *Mario*-like game, a long, flat platform punctuated by only coins and gaps may be playable and even provide an enjoyable challenge, but will likely not be visually interesting.

40.4.4 Engineering Constraints

Finally, there are important engineering constraints to consider when building a generator. Algorithm speed and efficiency are a big concern. While none of the approaches presented here are *unusable* for content generation, some are certainly better than others for specific domains. Simulation-based and evolutionary approaches tend to be fairly slow unless they are running alongside the player in real time. Constraint-based approaches are more difficult to predict. Depending on the number of variables and the kind of constraints that are expressed, constraint systems can actually solve some content generation problems quite quickly. Grammars usually generate content very quickly, but may need slightly more expensive generate-and-test loops to cull undesirable content.

There is also the concern of how long it will take to create the system and, importantly, test and debug it. Content generators are massively emergent, so bugs can be hard to find, even when following debugging practices such as generating from known random seeds. Constraint-based systems reduce concerns about whether the system will accidentally create unplayable content, but can be more difficult to author and debug. Most current constraint solving methods cannot tell you which combination of constraints is causing an unsatisfactory answer, though it is possible to build in debugging methods. On the other end of the spectrum, approaches that build content from relatively simple rules, such as simulations and grammars, do not make it so easy to make design guarantees but can be easier to author.

40.5 Tuning and Debugging a Content Generator

Because of their emergent nature, debugging PCG systems usually requires more than simply spot-checking different pieces of content that are produced. Standard practices for debugging systems that use a lot of randomness apply, for example, keeping track of random seeds so that bugs can be reproduced more easily and logging decisions made during generation (if possible, given the method used) to get an explanation from the system of what it's doing. For complex PCG systems, however, this alone is not enough to get a sense for what the system is doing and how changes made to the generator influence the content being produced.

Expressive range refers to the shape of the space of content that a PCG system can produce, as well as how easily that space can be controlled [Smith, G. 10]. A good method for understanding expressive range is to define important, measurable qualities (*metrics*) of the content being generated and produce a graph or other visualization of those metrics to see what kind of content is being created. By generating a large amount of content and evaluating each piece using those metrics, it is possible to get a view of the generator's performance. A first step is to look just at the minima, maxima, and medians for each metric—this will show examples of content that are at the extremes of the generator's abilities. For example, this is a quick way to see what the *easiest* and *hardest* levels are (for some approximation of difficulty), as well as the level at average difficulty. The metrics can be simple approximations, such as assigning scores to different level components based on their difficulty and averaging them together. More sophisticated metrics, including automated playtesting techniques [Salge 08], are also possible.

We can also use expressive range analysis to see if the generator seems biased to making any particular kinds of content by looking at the data in a histogram. Are there some bins that have more content in them than others? The generator may be biased toward creating that content over other kinds. A heatmap visualization can be applied to a 2D histogram to visualize *hot spots* in the generator and quickly see changes in the expressive range from different versions of the generator.

By investing in creating a system like this up front, it is possible to see how small changes to the generator lead to changes in the qualities of the content, be rerunning the metrics and seeing new expressive range charts. Expressive range can also be used to compare different generators [Horn 14].

40.6 Conclusion

This chapter has given an overview of techniques for content generation and discussed their trade-offs, described the many roles that PCG can have in a game and how to use those roles to guide decisions about which technique is best, and given some advice for getting started and for debugging your PCG system. This is only an overview of the field of PCG, and there are many more resources available for learning specific techniques or finding how other games have implemented their own content generators. Consequently, this chapter will close with a list of resources to point you toward more information.

40.6.1 Tools and Frameworks

While most off-the-shelf tools are not appropriate for use in large game projects, these tools and frameworks can be used as a prototype for your generator, to experiment and see what is possible:

- *Context Free Art* [Context Free Art 2014] and its cousin *StructureSynth* [Christensen 10] are accessible and well-documented tools for using shape grammars. They are intended for use by digital artists, but can be used to mock-up game content generators.
- *Choco* is a free and open source Java-based numerical constraint solver [Choco Team 2008].
- The *Potassco* suite of tools, particularly *clingo*, is a good tool for getting started with answer set programming [Gebser 2011].

40.6.2 Reading and Community

- *Procedural Content Generation in Games: A Textbook and an Overview of Current Research* is a book with chapters written by prominent researchers in the PCG community [Shaker 14].
- The PCG wiki is a community effort to create a central repository for articles and algorithms [PCG Wiki 14].
- The PCG Google group is an active community of developers and academics who share an interest in PCG [PCG Group 14].
- There is a great deal of academic research into PCG. Common venues for PCG research are the *Foundations of Digital Games* (FDG) conference, the *Procedural Content Generation* (PCG) workshop, and the *Artificial Intelligence in Interactive Digital Entertainment* (AIIDE) conference. Papers from AIIDE are available for free online through AAAI's digital library. Many FDG and PCG papers are archived in the ACM digital library.

References

[.theprodukkt 04] .theprodukkt. 2004. .kkrieger (PC Game).

[Adams 15] Adams, T. 2015. Simulation principles from Dwarf Fortress. In *Game AI Pro²: Collected Wisdom of Game AI Professionals*, ed. S. Rabin. A K Peters/CRC Press, Boca Raton, FL.

[[adult swim] games 10] [adult swim] games. 2010. Robot unicorn attack (PC game).

[Bay 12 Games 06] Bay 12 Games. 2006. Slaves to Armok: God of Blood Chapter II: Dwarf Fortress (PC Game).

[Bjork 04] Bjork, S. and Holopainen, J. 2004. *Patterns in Game Design* (*Game Development Series*), 1st edn. Charles River Media, Hingham, MA.

[Braben 84] Braben, D. and Bell, I. 1984. Elite (BBC Micro). Acornsoft.

[Choco Team 08] Choco Team. 2008. Choco: An open source java constraint programming library. *White Paper, 14th International Conference on Principles and Practice of Constraint Programming, CPAI08 Competition*, Sydney, New South Wales, Australia.

[Christensen 10] Christensen, M. H. 2010. StructureSynth. http://structuresynth.sourceforge.net (accessed June 22, 2014).

[Context Free Art 14] Context Free Art. 2014. Software. http://www.contextfreeart.org (accessed June 22, 2014).

[Dormans 10] Dormans, J. 2010. Adventures in level design: Generating missions and spaces for action adventure games. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (*Co-located with FDG 2010*), Monterey, CA.

[Ebert 03] Ebert, D. 2003. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, San Francisco, CA.

[Gearbox Software 09] Gearbox Software, and Feral Interactive. 2009. Borderlands (XBox 360). 2K Games.

[Gebser 11] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Schneider, M. 2011. Potassco: The potsdam answer set solving collection. *AI Communications* 24(2): 105–124.

[Hastings 09] Hastings, E., Ratan, J., Guha, K., and Stanley, K. 2009. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games* 1(4): 245–263. doi:10.1109/TCIAIG.2009.2038365.

[Horn 14] Horn, B., Dahlskog, S., Shaker, N., Smith, G., and Togelius, J. 2014. A Comparative evaluation of procedural level generators in the mario ai framework. *Proceedings of the Foundations of Digital Games 2014*, Fort Lauderdale, FL.

[Hullett 10] Hullett, K. and Whitehead, J. 2010. Design patterns in FPS levels. *Proceedings of the 2010 International Conference on the Foundations of Digital Games* (*FDG 2010*), Monterey, CA.

[Hunicke 04] Hunicke, R., LeBlanc, M., and Zubek, R. 2004. MDA: A formal approach to game design and game research. In *Proceedings of the 2004 AAAI Workshop on Challenges in Game Artificial Intelligence*. San Jose, CA: AAAI Press.

[Jennings-Teats 10] Jennings-Teats, M., Smith, G., and Wardrip-Fruin, N. 2010. Polymorph: A model for dynamic level generation. *Proceedings of the Sixth Conference on Artificial Intelligence for Interactive Digital Entertainment*, Palo Alto, CA.

[Koster 04] Koster, R. 2004. *A Theory of Fun for Game Design*, 1st edn. Paraglyph Press, Scottsdale, AZ.

[McGuinness 12] McGuinness, C. 2012. Statistical analyses of representation choice in level generation. In *2012 IEEE Conference on Computational Intelligence and Games (CIG),* Granada, Spain, pp. 312–319. doi:10.1109/CIG.2012.6374171.

[MicroProse 91] MicroProse. 1991. *Sid Meier's Civilization (PC Game)*.

[Müller 06] Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Van Gool, L. 2006. Procedural modeling of buildings. *ACM Transactions on Graphics* 25(3): 614–623.

[Onuczko 05] Onuczko, C., Cutumisu, M., Szafron, D., Schaeffer, J., McNaughton, M., Roy, T., Waugh, K., Carbonaro, M., and Siegel, J. 2005. A pattern catalog for computer role playing games. In *Game on North America*, Montreal, Canada, pp. 33–38.

[PCG Group 14] Procedural Content Generation Google Group. 2014. https://groups.google.com/forum/#!forum/proceduralcontent (accessed June 22, 2014).

[PCG Wiki 14] The Procedural Content Generation Wiki. http://pcg.wikidot.com/ (accessed June 22, 2014).

[Persson 11] Persson, Marcus. 2011. Minecraft (PC Game).

[Rogue Basin 12] Rogue B. 2012. Articles on implementation techniques. http://roguebasin.roguelikedevelopment.org/index.php?title=Articles#Implementation (accessed April 28, 2014).

[Salge 08] Salge, C., Lipski, C., Mahlmann, T., and Mathiak, B. 2008. Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games. In *Proceedings of the 2008 ACM SIGGRAPH Sandbox Symposium on Video Games (Sandbox'08)*, pp. 7–14. Los Angeles, CA: ACM.

[Saltsman 09] Saltsman, A. 2009. Canabalt (PC Game). Adam atomic. http://www.adamatomic.com/canabalt/ (accessed August 14, 2014).

[Shaker 14] Shaker, N., Togelius, J., and Nelson, M. 2014. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer. http://pcgbook.com.

[Smith, A. 10] Smith, A. M. and Mateas, M. 2010. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG),* Copenhagen, Denmark.

[Smith, A. 11] Smith, A. M. and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3): 187–200.

[Smith, A. 12] Smith, A. M., Andersen, E., Mateas, M., and Popovic, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. *Proceedings of the 2012 Conference on the Foundations of Digital Games*, Raleigh, NC.

[Smith, G. 11a] Smith, G., Anderson, R., Kopleck, B., Lindblad, Z., Scott, L., Wardell, A., Whitehead, J., and Mateas, M. 2011. Situating quests: Design patterns for quest and level design in role-playing games. In *Proceedings of the International Conference in Interactive Digital Storytelling,* Vancouver, British Columbia, Canada.

[Smith, G. 10] Smith, G. and Whitehead, J. 2010. Analyzing the expressive range of a level generator. *Proceedings of the Workshop on Procedural Content Generation in Games, Co-located with FDG 2010*, Monterey, CA.

[Smith, G. 11b] Smith, G., Whitehead, J., and Mateas. M. 2011. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, Special Issue on *Procedural Content Generation* 3(3): 201–215.

[Smith, G. 11c] Smith, G., Whitehead, J., Mateas, M., Treanor, M., March, J., and Cha, M. 2011. Launchpad: A rhythm-based level generator for 2D platformers. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 3(1), pp. 1–16.

[Smith, G. 12] Smith, G., Othenin-Girard, A., Whitehead, J., and Wardrip-Fruin, N. 2012. PCG-based game design: Creating endless web. In *Proceedings of the International Conference on the Foundations of Digital Games* (*FDG '12*), pp. 188–195. Raleigh, NC: ACM. doi:10.1145/2282338.2282375.

[Stiny 80] Stiny, G. 1980. Introduction to shape and shape grammars. *Environment and Planning B* 7(3): 343–351.

[Togelius 08] Togelius, J. and Schmidhuber, J. 2008. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games, 2008 (CIG'08),* Perth, Australia, pp. 111–118.

[Togelius 11] Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3): 172–186.

[Toy 80] Toy, M., Wichman, G., Arnold, K., and Lane, J. 1980. Rogue (PC Game).

[Tutenel 09] Tutenel, T., Smelik, R., Bidarra, R., and de Kraker, K. J. 2009. Using semantics to improve the design of game worlds. In *Proceedings of the Fifth Artificial Intelligence in Interactive Digital Entertainment Conference (AIIDE09)*, Palo Alto, CA.

[Yannakakis 11] Yannakakis, G. N. and Togelius, J. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2(3): 147–161. doi:10.1109/T-AFFC.2011.6.

[Yu 09] Yu, D. 2009. Spelunky (PC Game).