General Analytical Techniques For Parameter-Based Procedural Content Generators

Michael Cook*, Simon Colton*†, Jeremy Gow*, Gillian Smith‡
*Queen Mary University of London, United Kingdom
†Sensilab, Faculty of IT, Monash University, Australia
‡Worcester Polytechnic Institute, USA

***Abstract*—Most generative systems built in game development are parameter-driven, but the relationship between parameters and the output of the system is often unclear. This makes them frustrating to use for both experts and novices, and as a result generators are often filtered post-hoc, or tweaked through time-consuming trial and error. In this paper we introduce two analytical techniques: *smoothness* and *codependence*. We show how these features help analyse the impact of a parameter change on a generative system and suggest ways this could feed back into more intelligent tools that make working with procedural generators more precise and pleasant.**

***Index Terms*—procedural content generation, generative systems analysis, co-creativity**

I. Introduction

Procedural generation continues to grow and develop as a staple game development technique, both as a tool for use during development to accelerate and augment content creation, and a runtime game feature that can help achieve specific design goals such as surprise or unpredictability. Generative techniques can be seen at work in the biggest-budget AAA games as well as the smallest hobbyist creations, and we are seeing increasingly rapid adoption of experimental techniques (such as Wave Function Collapse [1]) in commercial titles (such as *Bad North* [2] or *Caves of Qud* [3]).

Despite its popularity, procedural generation has retained a reputation for being unpredictable, hard to control, and even lacking in artistry [4]. We argue that one of the reasons for this is a lack of good, standardised tools for interacting with and understanding generative software. This results in generative engineering being a fragmented practice, with individual developers building ad-hoc domain-specific tools, rather than having access to tools which bridge many domains and techniques, which would allow transferable skills and knowledge to develop and encourage a higher-level way of thinking and reasoning about generative systems.

Basic statistical analysis is sometimes employed by practitioners of generative software (e.g. measuring the rate a feature occurs in outputs), but few specialised tools and techniques exist to specifically analyse and help solve the problems inherent in building and using these systems. One of the better-known techniques is *expressive range analysis*, which helps provide a visual overview of a generative system's behaviour with respect to given metrics [5]. Several tools, such as *Danesh* [6], attempt to automate certain analytical tasks or invert common interaction processes to give the user more initiative in editing a generative system.

Most approaches to analysing procedural generators focus on evaluating the *output* of the system, reducing what Norman calls the *gulf of evaluation* [7], which is a measure of how difficult it is for a user to interpret the state of a system and to understand what changes they wish to effect upon it. In this paper we present techniques to complement this work, by providing ways to improve the user's understanding of the *inputs* to a generative system, and to provide more clarity over the actions currently available for them to take. Norman calls this the *gulf of execution*.

We present two analytical techniques which can be applied to parameter-driven generative systems: visualising the *smoothness* of a parameter, and the *codependence* of one parameter on another. Smoothness measures how consistent the impact of changing the parameter is on the generative system. Codependence measures the degree to which one parameter's smoothness is affected as the value of another parameter changes. We show how to calculate and visualise these, and give examples of how this analysis can reveal nuances in the behaviour of inputs to generative systems. We then discuss how these techniques can be embodied in new tools, both through explicit visualisation of the raw analysis, and through subtler user interface features that help novice users without overwhelming them.

The remainder of the paper is organised as follows: in §II and §III we introduce some background, related work, and introduce a running example; in §IV, §V and §VI we show how we measure a change in a procedural generator, and then go on to describe Smoothness and Codependence analyses; in §VII and §VIII we discuss how these techniques can be applied to tools, and detail future work; finally, in §IX we summarise our contributions.

II. Background

In [8] the authors provide a survey of search-based procedural generation, and identify in their taxonomy a distinction between 'random seeds' and 'parameter vectors', noting: "*At one extreme, the algorithm might simply take a seed to its random number generator as input; at another.. the algorithm might take as input a multidimensional vector of parameters.*"

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

We argue that most procedural generators implemented in games today are parameter-driven in some way or other. Even generators which rely heavily on seed-based generation, such as the noise functions which drive world generators in Dwarf Fortress or Minecraft, exist as part of a larger generative system which includes many parameters that control the interpretation or integration of that noise into a more complex generated artefact. Parameters allow both designers and players to control and edit the generator's output, even if the degree of control offered can be variable and confusing. For the remainder of this paper we will use the term 'generative system' or 'generator' interchangeably to refer specifically to any generative system with at least one non-seed parameter.

A. Definitions

In this paper we will discuss procedural content generation at different levels of abstraction, from single outputs to abstract multidimensional spaces of spaces. Here we define some terms that are useful for understanding the remainder of the paper.

We define a *procedural generator* as a piece of software with one or more inputs, called *parameters*. In this paper we assume these parameters are real-valued, as our analysis applies to parameters with an ordered range of values. We also follow [6] in assuming that a parameter has a lower and upper bound, either defining the limits of values it can take, or specifying a range of values that the user is interested in. When the generator's code is executed, it produces an *output* (sometimes called an 'artefact' or 'content').

A *parameterisation* of a generator is an instance of that generator with each of its parameters set to a particular value.

The *generative space* of a procedural generator as the set of all possible outputs a generator can produce under a particular parameterisation. For some generators this is simple to calculate, such as permutations of a set, e.g. dealing from a deck of cards. For other generators an upper bound is calculable, but the actual set of artefacts in the generative space may be smaller due to duplication during generation.

B. Related Work

Expressive range analysis is a technique proposed in [5] for analysing the specific features of the generative space of a generator. ERAs use functions called *metrics* which measure particular features of a piece of content, and use this to visually represent the system's generative space. In [5] the technique is applied to a procedural level generator for a platforming game, with metrics measuring *leniency*, which measures how forgiving a level is towards the player, and *linearity* which measures how variable the level's geometry is.

In [9] the authors present new visualisation techniques for expressive range that attempt to visualise it beyond two dimensions, helping a user look at multiple comparisons at once. They also introduce ways to further analyse the output of procedural generators, particularly for generative systems based on machine learning techniques. Like expressive range analysis, these methods focus on evaluating the output of generators, in this case finding new ways to characterise the quality of the generative space.

Many papers contribute specific evaluation criteria, such as [10], which evaluates a collection of generators which all create the same kind of content, and contributes new metrics for measuring the qualities of generated output in the domain of platforming games. The authors also discuss controllability in the paper, categorising the generators they assess based on how they are interacted with. Five of the seven generators surveyed are parameter-based.

Researchers have also tried to tackle the gulf of execution problem by developing techniques which are easier to control, such as in [11] where the authors present a constraint-based approach for controlling Markov-based generators. In [12] the authors introduce the notion of Procedural Content Generation via Machine Learning, or PCGML. One of the motivations for PCGML is that it '*avoids the complicated step of experts having to codify their design knowledge and intentions*' by allowing users to provide examples of content as training data for the PCGML system.

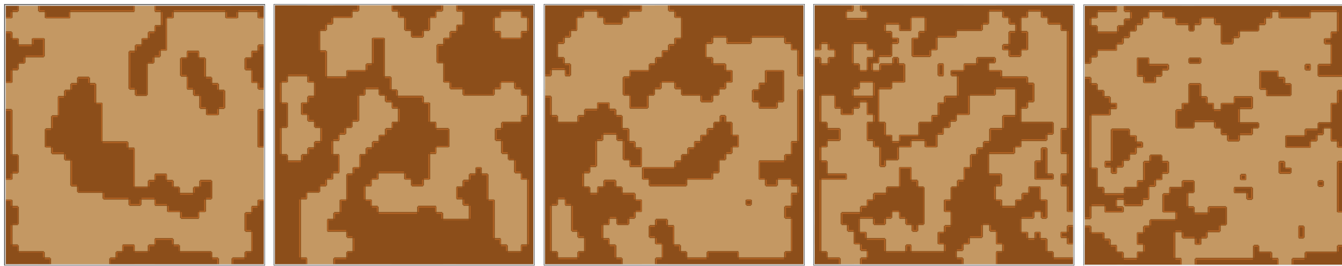
New techniques which do not rely on writing code, or allow different kinds of control over a generative system, are all valuable ways to advance procedural generation and make it more accessible. We see these approaches as complementary to our own, which is to make it easier to codify design knowledge and express design intent. Our aim is to help users build their understanding, confidence and ability with procedural generation through intelligent tools and analytical techniques.

III. Running Example

We will use a cellular automata-based generator as a running example throughout this paper. Cellular automata are popular among game developers working with procedural generation – it is one of only two tutorials about procedural generation provided by Unity [13], the world's most popular game development tool, and it has featured in many well-known games such as *Brogue*, *ADOM* and *Cogmind*.

The generator begins by creating a two-dimensional array. Each element in the array has a chance to be randomly set to either 1, denoting a solid tile, or 0, denoting an empty tile. The probability of this is controlled by the *Initial Solid Chance* (ISC) parameter. The generator then performs a number of passes on the array. The exact number of passes is controlled by the *Iterations* (ITR) parameter. In each pass it calculates the number of solid neighbours a tile has, n. It then applies two rules to each tile: 1) if the tile is solid and $n \leq x$, the tile becomes empty; and 2) if the tile is already empty and $n \geq y$, the tile becomes solid, where x and y denote the *Death Limit* (DL) and *Birth Limit* (BL) parameters, respectively. When the number of passes specified by the *Iterations* parameter is complete, the dungeon is rendered.

This generator, as well as a more complex variant with four additional parameters and two phases of generation, can be found as examples supplied with *Danesh* [14].



(a) ISC = 0.4, ITR = 6 (b) ISC = 0.5, ITR = 6 (c) ISC = 0.465, ITR = 6 (d) ISC = 0.465, ITR = 2 (e) ISC = 0.411, ITR = 2

Fig. 1: Five outputs from different stages of editing the parameters of a cellular automata-based dungeon generator.

A. Problem Scenario

To motivate our analysis techniques, we describe a scenario a game developer might encounter when working on a procedural generator. A generative system already has been built by another team member, which uses cellular automata to generate dungeons for the player to explore. Figure 1a shows an output from the system. The developer wishes to make the dungeons less open and more winding and cave-like.

They first come across the ISC parameter. Initially, $ISC{=}0.4$. They next set $ISC{=}0.45$. This improves the structure of the dungeons somewhat, and so they increment ISC by another 0.05 and set $ISC{=}0.5$ (fig. 1b). Suddenly, the caves are fragmented and unplayable. That is, the same small change had an unexpectedly more powerful impact on the system. After a lot of tweaking, they settle on $ISC{=}0.465$ as the value they want. Figure 1c shows a sample output at this stage.

They next decide to explore the impact of another parameter, ITR. Initially, $ITR{=}6$, which they reduce to $ITR{=}2$. This makes the walls more jagged and organic, which they are happy about, but this jaggedness now makes some of the dungeons disjointed, an example of which is shown in fig. 1d. To fix this, they return to the ISC parameter, reducing it from 0.465 to $ISC{=}0.45$. Even though this value produced good levels just a few moments ago, this slider is now behaving differently. They finally set the value to $ISC{=}0.41$ – almost as low as its original value of 0.4 – in order to obtain content they want. Figure 1e shows the final state of the generator. Despite finding a configuration they liked, the process involved a lot of guesswork and was hard to understand or predict.

In particular, this example identifies two problems: 1) As the value of a parameter changes, the nature of its effect on the output of the generator can shift unexpectedly, making it hard to predict the effect of changes based on past observations, e.g. when changing ISC from 0.45 to 0.5. 2) As the value of one parameter changes, it can change the way other parameters affect the output of the generator, e.g. when changing ITR from 6 to 2, the effect of setting ISC back to 0.45 was no longer desirable, forcing the user to make further changes.

IV. Measuring Change

The aim of our analysis is to provide insight into how input parameters affect the output of generative system. As such, an important part of our analysis is describing the difference between two parameterisations of a procedural generator – before and after a parameter change. Moreover, we wish to keep the analysis *general* in that it is agnostic to the kind of content being generated. We thus assume the existence of user-defined metric functions, in the same vein as expressive range analysis, to isolate important features of the generated content that we can evaluate.

Figure 2 shows two sets of sampled data from a cellular automata-based dungeon generator. The difference between the two sample sets is the value of the ISC parameter. In each case, the generator was sampled 250 times, and each sample was run through a metric which evaluated a property called *Connectedness*, which measures how accessible the open areas of the dungeon are from one another.

We identify two important features of generator samples such as these. The first is the *centroid*, which is the average metric score of the sample. The second is the *standard deviation* of the sample. In fig. 2, the centroid is marked with a red circle, and the red interval marks a distance of two standard deviations above and below the centroid (limited to the metric's value range of [0,1]).

The centroid and standard deviation give us a good understanding of the current sample distribution, showing the approximate center of the distribution and how dispersed the sample is, respectively. Throughout this paper we focus mostly on single-metric analyses, with no more than two parameters considered at any one time. This is to keep the visualisation simple; it is, however, straightforward to look at measuring centroids in multidimensional space to consider all metrics simultaneously, for example. We discuss the implications of this later in terms of visualising our analysis for the user.

We now proceed with a description of our two analysis techniques, which use this approach to measuring change, and illustrate our examples with our running example generator, the cellular automata-based dungeon generator.

V. Smoothness

The *smoothness* of a parameter describes how the impact of that parameter on the output of the generator changes across the range of values the parameter can take. In order to make visualisation and interpretation feasible, the smoothness analyses in this paper are done in two dimensions: comparing one parameter's value to the impact on a single metric.

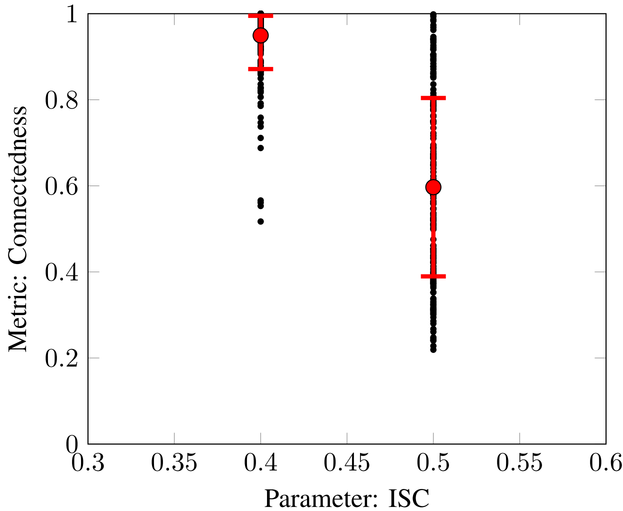


Fig. 2: A series of samples from a generator, using two different values for the ISC parameter. The x-axis records the ISC value used to generate the sample, and the y-axis records the *Connectedness* metric score of each sample.

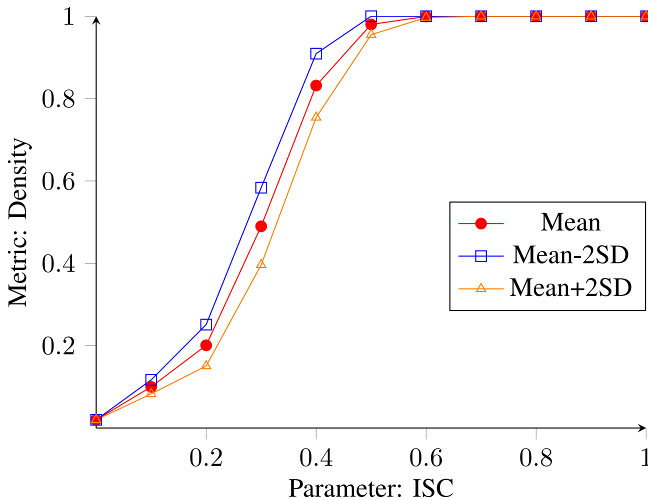


Fig. 3: A smoothness analysis of the ISC parameter of a cellular automata generator, measured against the Density metric.

To perform a smoothness analysis, we sample the parameterat regular intervals across the parameter's full range. This assumes the existence of minimum and maximum values, discussed in §II-A. For each value we then sample the generator many times. For the examples in this paper we sampled parameters at 10% intervals along their value range, and then took 250 samples from each parameterisation of the generator. Each sample is evaluated using the metric function relative to which we are analysing smoothness. We then calculate the average and standard deviation of the sampled metric scores.

Next, we create a two-dimensional plot showing each parameter value plotted against its average metric score, with additional points showing two standard deviations above and below the mean. Figure 3 shows a smoothness analysis of the ISC parameter of the cellular automata dungeon generator, sampled at 10% intervals, relative to the *Density* metric, which measures the percentage of solid tiles in a dungeon.

Studying the smoothness analysis can reveal insightful information about the behaviour of the parameter. The first question we can answer is whether this affects the metric in question at all: this is easy to see by looking at the highest and lowest mean metric values recorded. In fig. 3 we can see this stretches the full range of the metric from 0 to 1, meaning that this parameter has a strong impact on this metric in the current parameterisation.

To learn more about the behaviour of the parameter as its value changes we can study the gradient of each *segment* of the smoothness graph. A segment of the smoothness graph is the section between two adjacent sampled parameter values, representing changing the parameter from one value to another. Steep gradients indicate segments with large impacts on the metric scores, while shallower gradients indicate little or no impact. Moreover, a climbing gradient denotes that the parameter and metric are positively correlated, while a falling gradient denotes that the parameter and metric are negatively correlated. In fig. 3 we can see that segments between 0.6 and 1.0 have gradients close to zero, meaning they have little impact on the metric, while the segment between 0.3 and 0.4 has a gradient of 3.42, indicating a high impact.

The gradient of an individual segment indicates impact for that part of the parameter range, but we can also assess how predictable changing the parameter value is in general by assessing how much the gradient varies across the parameter range. A simple way to assess this is by calculating the standard deviation of the segment gradients. A low standard deviation means the parameter is generally smooth – there is little difference between the gradients of segments, indicating a generally straight-line graph. A higher standard deviation suggests the parameter behaviour is less predictable. In fig. 3 we can see that this parameter is indeed quite unpredictable. The lowest segment gradient is 0, while the highest is 3.42, with an overall standard deviation of 1.21. Changing the parameter from 0.1 to 0.2 has a much lower impact on the metric average than changing it from 0.3 to 0.4, despite this being the *same* amount of change to the input in both cases.

In addition to estimating predictability and magnitude of impact, we can also use domain-specific knowledge about our generator to intuit things such as meaningful ranges to explore. For example, in the case of fig. 3 we might know that outputs with a Density metric score of less than 0.2 or more than 0.8 are not usable, in which case we can restrict our exploration of parameter values to a range of 0.2 and 0.4. We can also use the standard deviation markers to compare the dispersion of the generator at different parameter values. For example, increasing a parameter might cause no change to the position of the average metric score, but might increase the standard deviation, meaning the output of the generator is much more dispersed according to this metric.

Smoothness analysis does not tell us how useful, meaningful or expressive a parameter is; instead, it allows us to assess abstract qualities of the parameter that relate to how impactful and predictable the parameter is. Note that smoothness analyses only hold on the specific parameterisation for which they are performed. This means that if the value of another parameter is changed, the smoothness analysis must be recalculated. While this is a quick solution and fixes the immediate problem, it doesn't provide the user with an understanding of why the smoothness has changed, or what the implications are if other parameters change again in the future. To solve this problem, we propose *codependency analysis* in the following section.

VI. Codependence

A *codependency analysis* of two or more parameters shows how the value of one parameter impacts the smoothness of the others, for a given metric. In this way, codependency can be thought of as a higher-dimensional smoothness analysis, but unlike smoothness its function is to establish the relationship between parameters, rather than illuminate the behaviour of a single parameter. Many parameters in generative systems are orthogonal in their impact on the system – changing the value of one has no effect on the value of the other. However, some parameters have complex relationships with one another. For example, as we demonstrate shortly, in our running example of the cellular automata dungeon generator, the Birth Limit (BL) and Death Limit (DL) parameters are highly codependent on one another. Any change to one of these parameters impacts how the other behaves.

Our basic smoothness analysis which we introduced in the previous section was two-dimensional, featuring one parameter and one metric. As we introduce a second varying parameter to the analysis here, the basic codependency analysis is three-dimensional. Given two parameters $p1$ and $p2$ and a given metric m, we carry out a codependency analysis of $p1$ and $p2$ against m as follows. First, we sample parameter $p1$ at regular intervals along its range (as with smoothness, we choose 10% intervals for the examples in this paper). Next, for each value of $p1$ we perform a smoothness analysis for parameter $p2$ against the given metric m. This produces a series of smoothness analyses, which can be plotted on three axes as a surface. Note that codependency analysis is symmetric, meaning that the same analysis can be performed regardless of the ordering of $p1$ and $p2$.

For example fig. 4a shows a codependency analysis of the ISC and the ITR parameters from the cellular automata generator, against the Openness metric (i.e. $p1$ = ISC, $p2$ = ITR, and m = Openness). The openness metric measures what proportion of empty space in the dungeon is entirely surrounded by other empty space. As shown in fig. 4a, although the ISC parameter has a significant impact on the Openness score, changing the value of the ITR parameter generally does not affect the way ISC behaves, except at very low values. When the value of the ITR parameter is low, the behaviour of the ISC parameter changes drastically. Figure 5a shows three slices from the codependency plot, which clearly shows the impact of a low ITR value.

As another example, fig. 4b shows a codependency analysis of the Birth Limit (BL) and Death Limit (DL) parameters, against the Density metric, from the same generator (i.e. $p1$ = BL, $p2$ = DL, and m = Density). Here we can see a broader codependency relationship between the two parameters – almost anywhere on the plot, if one parameter is changed, the smoothness of the other parameter changes too. However, the sharpness of the change is not as intense as the example in fig. 4a. We can see evidence of this in fig. 5b, which plots three slices from fig. 4b. As shown, the slices are more spaced apart, but the general shape of their curve is closer to each other than the slices in fig. 5a. In both cases there is evidence of codependency, but it takes different forms: in fig. 4a codependency is isolated to a small part of the parameter range; in fig. 4b codependency is spread across the range but smaller in magnitude.

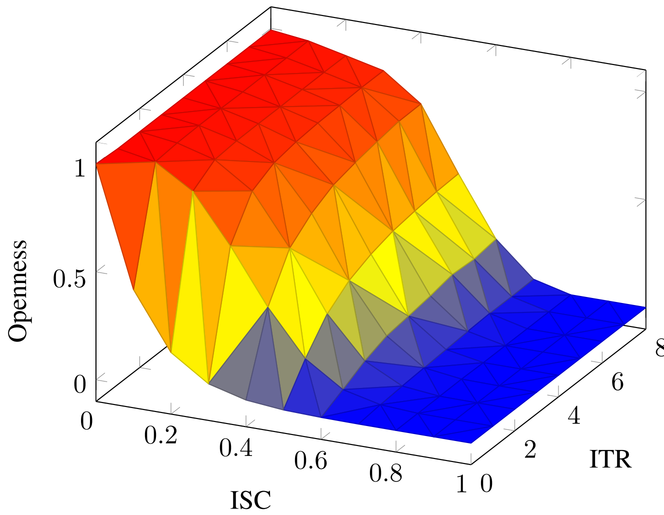
VII. Example Applications in Danesh

Smoothness and codependency analyses are useful for those engineering generative systems. They can be used to reveal hidden relationships between inputs, and both to solve and to express problems experienced by users editing and using generative systems. These techniques can also be implemented within domain-agnostic tools to automate analysis for non-expert users. This not only makes the techniques more accessible and easier to perform, but it can also enable new forms of interaction that allow novice users to benefit from the insight of these analytical techniques without having to interpret them directly. To this end, we implemented both our smoothness and codependency analyses in Danesh, a tool for analysing procedural generators [6]. Using our smoothness analysis in Danesh enabled us to develop a technique we call *automated parameter smoothing*, which makes interaction with parameters more natural, and to prototype *codependence highlighting* as a subtle indicator of parameter interrelation.

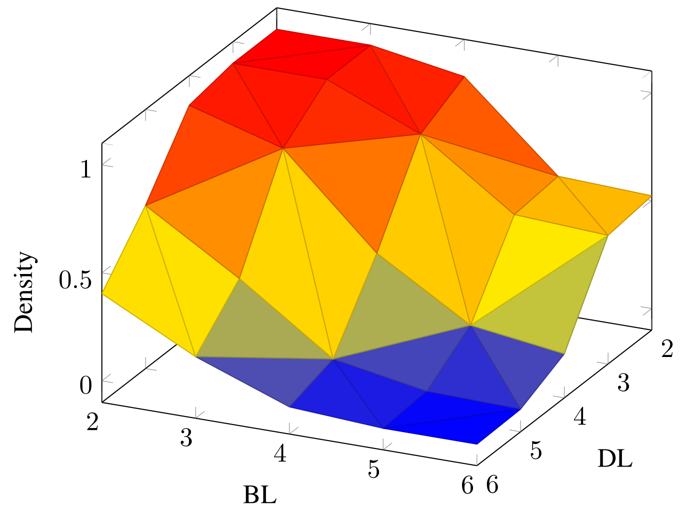
A. Danesh

Danesh is an interactive tool for exploring, explaining and experimenting with generative systems [6]. The current version of Danesh is written as a plugin to the Unity game development environment. Danesh is designed to help both novice and expert users accomplish a variety of tasks related to generative software, from viewing output from the system or changing parameters, through to more complex operations such as performing randomised expressive range analyses, or automatically searching the parameter space to find a target configuration for the generator in the expressive space [15].

Danesh is designed to be content-agnostic, so it can operate on any generative system. Users write their own metric functions that describe properties of the content they are interested in exploring, which Danesh uses to process and analyse content, including when it applies automatic analysis techniques such as expressive range analysis, or automatic parameter optimisation [15]. When a generator is loaded into

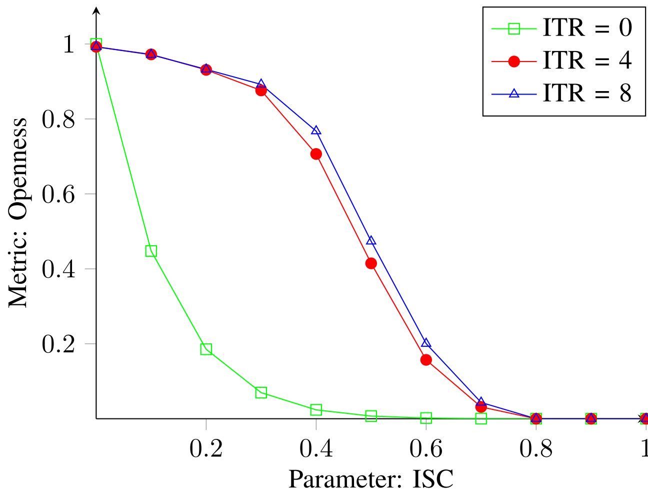


(a) A codependency analysis of the Initial Solid Chance (ISC) and Iterations (ITR) parameter, against the Openness metric.

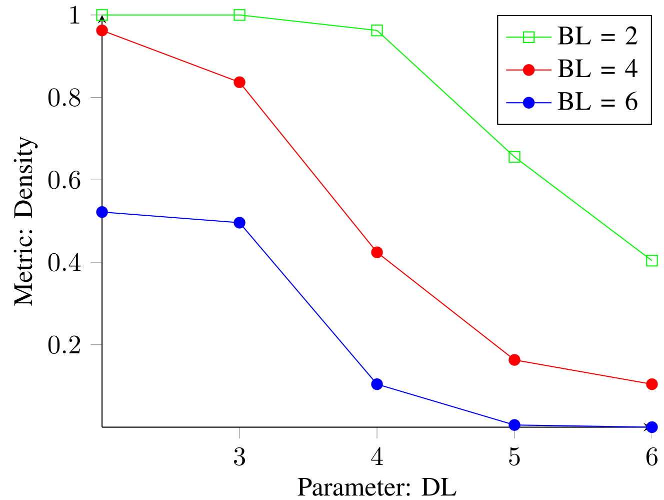


(b) A codependency analysis of Birth Limit (BL) and Death Limit (DL), against the Density metric.

Fig. 4: Two codependency analyses for the cellular automata dungeon generator.



(a) Three slices from fig. 4a.



(b) Three slices from fig. 4b.

Fig. 5: Smoothness plots showing slices through the codependency graphs at particular parameter values.

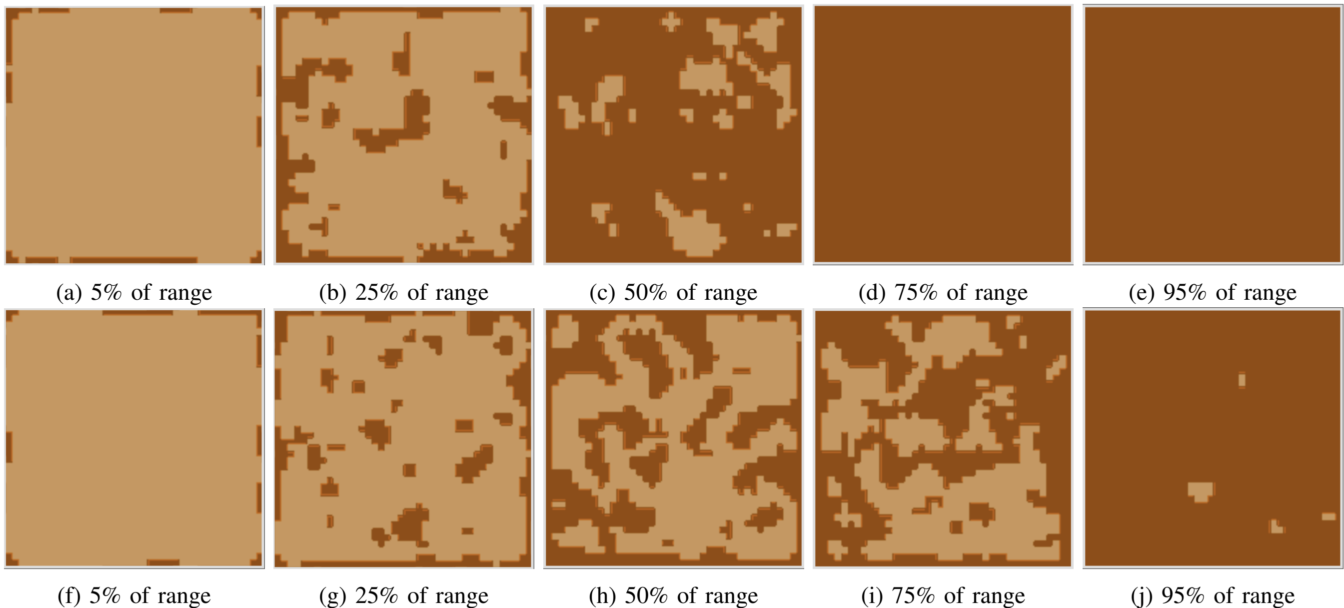
Danesh, the tool scans the code for annotations that mark parameters the user wishes to interact with, along with user-specified maximum and minimum values. These parameters are then displayed on one of the tool's tabs, with sliders capped at the maximum and minimum values.

B. Automatic Parameter Smoothing In Danesh

As we described earlier in this paper, parameters that do not exhibit smooth behaviour can be unpredictable and hard to control. Figure 3 shows an example of this, with a varying gradient of change across the range of the parameter's values. An ideally smooth parameter would not exhibit this behaviour – instead, its smoothness graph would be a straight line with a gradient of 1, as in the graph of $y = x$. In this case of perfect smoothness, the change in metric score is consistent and monotonic throughout the full range of the parameter, and thus much easier to control and think about.

Smoothness analysis provides a lot of insightful information, but this may be overwhelming for less experienced users, and it does not provide a direct way to fix unsmooth parameters. To solve this, we have implemented a technique called *automatic parameter smoothing* in Danesh, which transforms the controls for a parameter so that they appear to the user to have a more linear relationship with a target metric.

For instance, consider the outputs from the dungeon generator in the top row of fig. 6 as the ISC parameter is increased along its value range. These outputs are unsmooth with respect to the Density metric, which we can see from



(a) 5% of range (b) 25% of range (c) 50% of range (d) 75% of range (e) 95% of range

(f) 5% of range (g) 25% of range (h) 50% of range (i) 75% of range (j) 95% of range

Fig. 6: Sample outputs from an unsmoothed (top) and smoothed (bottom) parameter value range.

how rapidly the density changes in some parts of the range, and how slowly it changes in others. Between 25% and 50% of the parameter's max value, the density soars from 20% to over 95% (which we can verify by cross-referencing fig. 3). This means that 75% of the range of density values the parameter can cover are expressed through just 25% of the parameter's range. To address this, we can automatically smooth the parameter, to allocate a wider bandwidth to more meaningful parts of the parameter's value range. An example of a smoothed parameter is shown in the bottom row of fig. 6. We can see that the values between 25% and 75% now express a more useful range of outputs. In fact, the central 50% of the smoothed parameter range expresses just 13% of the original, unsmoothed parameter's range, showing how impactful the effect of smoothing can be.

To perform automatic smoothing on a parameter, p, with respect to a metric, m, we first compute a smoothness analysis to derive a graph similar to fig. 3. We note the lowest and highest recorded average metric score as m_{min} and m_{max} respectively. This will be used to smooth the user's input in the next step. We now replace the unsmoothed parameter slider with a new smoothed parameter slider. This slider looks the same, but has no markings for minimum or maximum value, instead abstracting the range of the parameter. Internally, this slider is mapped onto the range $[0, 1]$. When the user chooses a value on the smoothed slider, we map this value from $[0, 1]$ back onto the range $[m_{min}, m_{max}]$. This value represents the expected value for m under the smoothed parameter, which we call m_{exp}. To find the corresponding value of parameter p, we look up m_{exp} on the original smoothness analysis, and interpolate a value for p that corresponds to m_{exp}.

Smoothed parameters must currently be requested manually by the user, because they are dependent on the current parameterisation of the generator and relative to a particular metric. The user picks a metric they wish to analyse smoothness in relation to, and the smoothed parameter can then be used until any other parameter is changed. At this point, the smoothed parameter is replaced by the original unsmoothed parameter until a new smoothness analysis is performed. The more intervals the underlying smoothness analysis is sampled at, the smoother the resulting parameter is.

C. Codependence Highlighting

We have implemented automatic codependency analysis into Danesh, and have begun prototyping a way for it to automatically highlight parameters that are affected by a change as the user edits parameter values. To do this, we calculate codependencies in advance between all parameters in a system, and then when a user begins to change a parameter p from one value to another, for each other parameter q we take two slices across the codependency surface, for the old and new values of p. We calculate the difference between the two slices, and use the magnitude of the difference to recolour the name of the parameter in Danesh's interface. A mockup of this is shown in fig. 7. We are still exploring the best way to reduce the difference between two slices to a scalar value, and so will continue to develop this feature in the future.

VIII. Future Work

1) Non-monotonic Parameter Smoothing: Smoothness analysis can sometimes reveal that parameters have non-monotonic relationships with metrics. This causes problems for our parameter smoothing approach, since it makes our inverse lookup technique impossible as it is no longer operating on a function (if m both increases and decreases as p increases, then there are some values of m which are obtainable from more than one value of p).

Fig. 7: Highlighted parameters after codependency analysis.

To overcome this, we believe we can partition the smoothness analysis into multiple segments by dividing it at every *stationary point* on the line. Each segment of the analysis can then be smoothed on its own, using the same process as described earlier, with a slight adjustment to the limits for the parameter value in each segment. This provides us with $n + 1$ smoothed intervals for n stationary points in the line. We propose labelling or recolouring these segments of the smoothed parameter to make it clear to the user that these regions exhibit different behaviour, even after smoothing.

2) N-Dimensional Analysis: Throughout this paper we have focused on two- and three-dimensional analysis of generative systems. This level of dimensionality is useful as it is easily illustrated and thus communicated to others, and is also of a reasonable dimensionality for someone to be able to understand and reason about.

One of the problems with this kind of analysis, however, is that only part of the system is being examined at any one time. It is quite possible to calculate a smoothness analysis of a parameter against all metrics in a system simultaneously, but this results in a multi-dimensional shape that is extremely hard to render legibly or imagine. Merely thinking about this system, much less performing mathematical operations or comparisons on it, is difficult for humans. However, it is not any harder for software, and in fact it may be preferable for a system such as Danesh to analyse all dimensions of this problem simultaneously. A point of future work is to assess what new automated operations become available by performing an n-dimensional analysis of the relationships between metrics and parameters for a given system, and whether existing techniques (like codependency highlighting on parameters) become more meaningful or accurate when extended to n-dimensions.

IX. Conclusions

Parameter-driven approaches to procedural content generation remain a popular technique for generating content for games. Their natural structure as a system with adjustable controls which lead to changes in output makes them appealing for newcomers, which makes it likely they will remain a popular technique for some time. As such, it's important that we develop a robust set of general analytical techniques for describing the behaviour of these systems. This is important not only so that we can formally discuss them, but also so that we can build better tools that compensate for the more confusing, complex or unclear aspects of these systems.

Many approaches to analysing procedural generators thus far have focused on understanding the output of the generator, such as ERAs. In this paper we focus on the other half of the system, to try to provide ways to analyse how the parameters that control a procedural generator impact the behaviour of that generator. We introduced two new terms: *smoothness*, to describe the linearity and monotonicity of a single parameter's impact on the output of a generator; and *codependence*, to describe how the smoothness of a parameter is affected by the value of another parameter. We showed how to calculate and represent these properties using sampling techniques. We also showed how these techniques can be applied to tools, by showing how a nonlinear parameter-metric relationship can be linearised and presented to the user in a different way.

Building better tools that are easy to use and offer powerful new modes of expression is vital for advancing procedural content generation, and in particular its role as a tool for game designers, not just technical programmers. The field of generative software research and practice has suffered for decades from severe fragmentation, particularly within games where neither code nor knowledge seem regularly transferrable. We believe that pushing for more general analytical techniques, and finding ways to embed them into tools which are accessible and understandable, is one of the best things we can do to advance the field.

X. Acknowledgements

The first author is supported by the Royal Academy of Engineering under the Research Fellowship scheme.

References

[1] ExUtunmno, "Wave function collapse," https://github.com/mxgmn/WaveFunctionCollapse.
[2] Plausible Concept, "Bad North," Nintendo Switch, 2018.
[3] Freehold Games, "Caves of qud," 2015.
[4] G. Duncan, "No Man's Sky: How I Learned to Love Procedural Art," https://www.youtube.com/watch?v=vcEA41eBOGs.
[5] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the Workshop on Procedural Content Generation in Games*, 2010.
[6] M. Cook, J. Gow, and S. Colton, "Danesh: Helping bridge the gap between procedural generators and their output," in *Procedural Generation Workshop, FDG*, 2016.
[7] D. A. Norman, *The Design of Everyday Things*. Basic Books, Inc., 2002.
[8] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," 2011.
[9] A. Summerville, "Expanding expressive range: Evaluation. methodologies for procedural content generation," in *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2018.
[10] S. Dahlskog, B. Horn, N. Shaker, G. Smith, and J. Togelius, "A comparative evaluation of procedural level generators in the mario ai framework," in *Proceedings of the International Conference on the Foundations of Digital Games*, 2014.
[11] S. Snodgrass and S. Ontañón, "Controllable procedural content generation via constrained multi-dimensional markov chain sampling," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 2016.
[12] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgrd, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, 2018.
[13] Unity, "Cellular automata," https://tinyurl.com/unitycellular.
[14] Unity Asset Store, "Danesh," https://tinyurl.com/daneshdl.
[15] M. Cook, J. Gow, and S. Colton, "Towards the automatic optimisation of procedural content generators," in *IEEE Conference on Computational Intelligence and Games*, 2016.