Danesh: Interactive Tools for Understanding Procedural Content Generators

Michael Cook, Jeremy Gow, Gillian Smith, and Simon Colton

***Abstract*—In order to advance the field of procedural content generation, and transfer knowledge from academic research to everyday use, we need to develop tools that make generative systems easier to understand and control. In this article, we introduce Danesh, a plugin to the unity game development environment, which helps provide a suite of tools that provide automation or analysis of different aspects of procedural generators. We describe here the features of Danesh, including automatic analysis of generated content, the visualization of generative spaces, automatic parameter discovery, and interface smoothing. We also provide reflections on our development of the tool so far.**

***Index Terms*—Computational creativity, generative software analysis, procedural generation.**

I. Introduction

PROCEDURAL generation is the application of generative algorithms to the production of content for games. Procedural generators are an important part of the landscape of modern game development, not only the reserve of programmers but also used by artists [1], musicians [2], writers [3], and people in many other roles. Despite being in common use among game developers, attitudes toward procedural generation are often negative, with many games now explicitly marketing themselves as including "hand-crafted" content. In his 2015 GDC talk Grant Duncan, Art Director on *No Man's Sky*, which extensively used procedural generation, said that he felt skeptical toward the idea initially and many of his peers told him procedural generation would "take control away from artists" and produce "endless, boring" content [4].

Grant also cites the confusion and lack of understanding among gaming audiences, too: "after announcing *No Man's Sky*, if you read the comments [on online articles] it turns out that nobody seems to know what procedural generation actually means." Duncan goes on to explain that procedural generation is "a big box of maths." The description of procedural generation as "maths" was common throughout the development of the game, both in the PR and the subsequent press coverage [5].

This ironic detachment highlights a major problem with procedural generation in the modern games industry: although prevalent, powerful, and popular, procedural generation is not well understood and this contributes to negative preconceptions. We believe that this feeling of a lack of control, and the perception that procedural generators are a dark art of "maths," is caused in part from the way that procedural generators are commonly presented. Generative algorithms are often shown as black boxes, with mysterious input parameters that behave inconsistently, and outputs that vary wildly and have unknown distributions. In addition to this, a common language for discussing and thinking about procedural generators has been slow to develop. This is due to the field being heavily fragmented—music generators are not spoken about in the same context as level generators, for example. It is also exacerbated by a lack of code reuse between projects or developers.

To attempt to tackle some of these problems, we have developed a tool called Danesh. Danesh is an open-source plugin for unity, one of the most popular game development tools that allows procedural generators to be viewed, edited, and analyzed in a single unified interface. It requires minimal setup to work with a generator, and can work with any kind of content (as long as the user can perform some simple setup). Danesh offers simple features such as editing inputs and viewing outputs; as well as more complex operations such as viewing the distribution of a generator's outputs, or automatically searching the parameter space for configurations that produce a specific outcome. Danesh has been presented at the 2017 Game Developers Conference, the largest industry event in the world, as well as used as a teaching tool in classrooms.

In this article, we outline the current features of Danesh, from simple tasks such as loading and viewing generators through to more complex analytical techniques. In doing so we also introduce for the first time automatic parameter discovery, a new technique for searching a code library to automatically suggest possible inputs to a generative system. In addition to this system description, we also reflect on our experience of building a tool aimed at game developers. We conclude by looking at the future paths for Danesh's development.

II. Related Work

A. Assistive Interfaces for Generative Systems

Several tools exist which attempt to provide assistance in designing a procedural generator. Tanagra [6] is example of

Manuscript received 20 April 2020; revised 25 September 2020; accepted 27 October 2020. Date of publication 7 May 2021; date of current version 15 September 2022. The work of Michael Cook was supported by the Royal Academy of Engineering Research Fellowship Scheme. *(Corresponding author: Michael Cook.)*

Michael Cook, Jeremy Gow, and Simon Colton are with the Queen Mary University of London, E1 4NS London, U.K. (e-mail: mike@possibilityspace.org; jeremy.gow@qmul.ac.uk; simon.colton@qmul.ac.uk).

Gillian Smith is with Worcester Polytechnic Institute, Worcester, MA 01609 USA (e-mail: gmsmith@wpi.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TG.2021.3078323.

Digital Object Identifier 10.1109/TG.2021.3078323

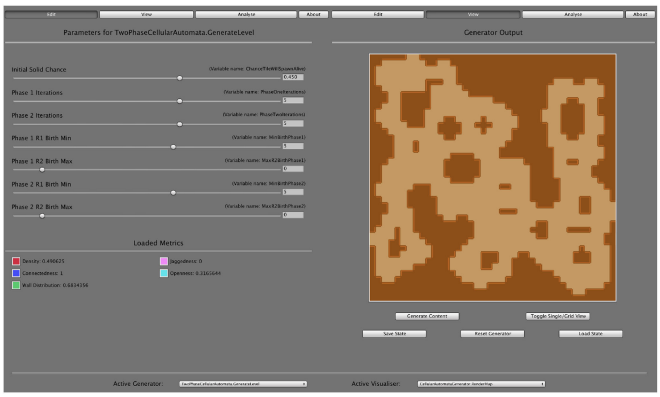

Fig. 1. Screenshot of Danesh's main user interface, showing parameter controls (left) and an example output (right).

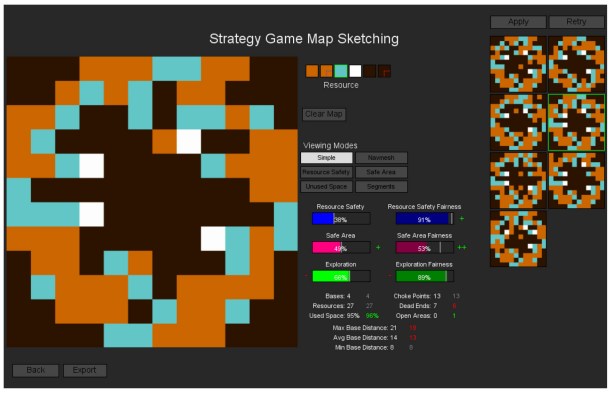

Fig. 2. Screenshot of the Sentient Sketchbook's interface. Alternative outputs are on the right, metrics and analysis are in the central area.

early research in assistive tools for generator design. Tanagra allows the user to design part of a platformer level, expressing constraints on the remainder of the design, which the tool then attempts to fill in. The user can go on to respond to this or see other output from the system.

The Sentient Sketchbook [7] is a tool for designing 2-D spaces for a variety of game contexts, such as real-time strategy game levels, or world maps for a role-playing game or story. The Sentient Sketchbook has an innovative user interface, which allowed the user to move between detail levels, thus allowing for high-level constraint expression at a low resolution, before moving in to examine possible generated content at higher detail. The Sentient Sketchbook also recalculates and surfaces metric scores while the tool is being used, allowing the user to appreciate differences between potential levels that may not be immediately obvious, as well as constrain along these metrics. Fig. 2 shows a screenshot of the interface.

The abovementioned systems are examples of specific generative systems, rather than a general-purpose tool for analyzing systems. However, their designs offer useful indicators of how interaction with generative systems can be done. Primarily, these systems work by alternating phases of users adding constraints and then the system responding. This is a similar rhythm, which we aim for with Danesh, except it takes place at a meta-level, alternating phases of adjusting a space of content, and then visualizing the results of the changes.

B. Generative Software Analysis

Expressive range analysis (ERA) is a way of analyzing the behavior of a generative system by sampling its output and visualizing the distribution of various properties of the sample [8]. Smith *et al.* used this system to analyze the space of artifacts created by the Launchpad [9] generator, and how it varies based on varying input parameters to the generator. We explain this technique more in depth later in this article, as it is a central part of Danesh's capabilities. ERAs are often visualized as a 2-D histogram, however, other visualizations have been used including using clustering, single-property frequency charts, and studies of change over time.

Core to the ERA approach to evaluating generators is the definition of appropriate metrics. Canossa and Smith [10] propose several potential metrics for 2-D platforming games that move beyond linearity and leniency, as proposed in the original ERA paper, and include aesthetic and experience-based metrics. However, these metrics have not been operationalized in existing generative systems. In [11], the authors evaluate commonly employed metrics for analyzing platformer level generators. Writing good and insightful metrics is a crucial step in analyzing a procedural generator, but both the process and the metrics remain relatively underexplored. The authors conclude that metrics alone are not sufficient for understanding a procedural generator, but also suggests that they are useful in providing insight.

In [12], Summerville proposes new ways of analyzing procedural generators that can visualize higher dimensions of data or provide different kinds of insight to ERA. Corner plots are suggested as a visualization technique, able to contrast multiple 2-D plots simultaneously, overcoming a limitation of expressive range, which makes it hard to consider more than two dimensions at once. Summerville also proposes methods for assessing other aspects of a generative system, such as how often elements from a training set reoccur in the output (particularly applicable to the emerging subarea of PCGML [13]).

C. Explainable AI (XAI)

Danesh is a tool for making sense of complex systems and, thus, has much in common with research into XAI. In [14], the authors put forward a proposal for XAI for game designers (XAID), which suggests different ways of thinking about and building tools for cocreation in game design, and the challenges inherent in doing so. The proposed three axes of XAID—explainability, initiative, and domain overlap—are all relevant to the design of Danesh.

XAID is especially relevant in the face of a rising interest in machine learning, which presents many challenges in education and training. In [15], the authors present tools for visualizing different kinds of information about the state of a machine learning system, in the context of a game design task. We share similar goals in the creation of Danesh, in that our intention is to create a rich tool that can layer in additional information as the

user wishes, to help improve their understanding of a dynamic system.

III. System Overview

Danesh is a plugin for the unity game development environment, written in C#. Unity is one of the most popular tools in the world for developing games—50% of mobile games are developed with unity, according to their own statistics.[1] In addition to being popular among commercial game developers, unity is free to use, which also makes it a common choice for students, hobbyists, digital artists, and many more groups besides. This makes it an excellent platform to target in order to support a wide variety of generative software developers.

Our guiding philosophy in designing Danesh was to build a tool that could be customized easily to work with any generator. This meant developing the tool to be agnostic to the type of content being generated. As a result, Danesh has been designed with "context gaps"—minimal spaces where understanding about the domain can be supplied by the user. Currently this places a nontrivial burden on the user, as they must write code to visualize generated content, measure interesting properties, and identify important regions of the code. We discuss how we have begun to overcome these limitations later in this article, through a mix of intelligent systems that attempt to automate some of these processes. Overall, we believe that Danesh's domain-agnostic design is its greatest strength, and a good approach to building a tool designed to work with many types of generator.

A. Loading and Viewing Generators

In order to load a generator into Danesh, the user tags parts of their code with annotations. Annotations label parts of a codebase, allowing the labeled fields, methods or types to be discovered through C# metaprogramming. This allows Danesh to automatically configure itself without major refactoring or restructing of the game's codebase, instead simply asking the user to mark methods or fields that are relevant to the procedural generation.

In order to minimally run Danesh, the user must mark one method with the `Generator` annotation. When Danesh runs, it scans the codebase for a `Generator` annotation, and stores a reference to that method as the generator. If multiple generators are marked, Danesh allows the user to switch between generators while using the tool (the interface updates accordingly to show data and controls relevant to the current generator). In line with our agnostic approach, Danesh does not know what type of object is returned by the generator method. Instead, as we describe in this section, the user is asked to provide small code snippets to provide context-sensitive code to Danesh. An example of this is visualization. Danesh can be run without visualizers, but if the user wishes to see the output of a generator (which is recommended for content with a visual component) they can provide a visualizer method, which takes content output by the generator as a parameter, and returns a texture, which displays some kind of rendering of the content. As with the generator method, this is written in the codebase and tagged using an annotation (if the user does not wish to write a visualizer, they can use an alternate method, which displays the output of an in-game camera). With these two methods, Danesh allows the user to open the main Danesh window, generate content, and display it on the screen. We currently do not support audio outputs, but this is a point of future work.

With this information configured, the user can now load Danesh and use the generate tab, which lets them press buttons to generate and view content (using the discovered annotated methods). This can be done in bulk or single-shot generation runs. Fig. 1 shows the main Danesh interface with the generate tab on the right-hand side.

B. Viewing and Editing Parameters

After viewing content, the next most fundamental feature is editing inputs to the system. The user can annotate fields in the codebase with the `Tunable` annotation to indicate that a field is of interest as a parameter to the system. The `Tunable` annotation also allows the user to provide a name for the parameter, and a maximum and minimum value (if of a numeric type). Although some parameters do not necessarily have maximum or minimum values, some Danesh functions require a capped range of values, and we also require limits in order to display the parameter in the interface. We recommend the user enters a range they believe will be interesting to explore or, failing that, a range of values that is reasonable for the parameter to take (for example, not exceeding the maximum value seen for that parameter in the past).

Once Danesh finds a field marked with the `Tunable` annotation, it creates a slider on the interface that allows the user to directly edit the value of the field. Fig. 1 shows this basic interface, with sliders on the left panel.

C. Metrics

Many of the more complex analytical features in Danesh require insight into the properties of generated content. In order to obtain this information while also being agnostic to what content is being generated, we ask users to define metrics, functions that take a piece of content as an input, and return a floating-point value in the range [0,1] as output.

Metrics typically are defined to measure a property of the output that is interesting to the user, and often one which is related to inputs in an unclear way. For example, if a parameter for a level generator directly affected the chance of spawning treasure, then a metric for measuring the amount of treasure in a level might not be that interesting—the user can intuit this fairly easily from the value of the treasure parameter. However, a metric that measures treasure "safety"—proximity to enemies, traps, or distance from the exit—might vary based on other factors in the generative system and, thus, be more interesting to measure and visualize.

Danesh supplies several example metrics as part of its default installation, designed to work on the example generators that are also supplied with it. These metrics can be easily repurposed for common content generation tasks too (such as 2-D grids).

[1][Online]. Available: https://unity3d.com/public-relations

Expert users may have little trouble identifying features of their content they wish to analyze and writing metrics to describe them. However, this process is harder for novice users, especially users who may not know how to program—we discuss this in future work.

Danesh cannot assess the quality or suitability of metrics defined by the user. For example, if a metric takes a long time to compute, the overall performance of Danesh will suffer. Similarly, if the calculation of a metric is affected by randomness or noise, this might reduce the reliability of any samples. We discuss in future work how we intend to improve and automate some aspects of the process of defining metrics.

D. Cellular Automata Example

In order to illustrate different aspects of Danesh, throughout this article, we use a simple cellular automata-based generator, which creates 2-D images that can be used to represent game levels, art, or other content. Our implementation is based on [16]. We configured our implementation for integration with Danesh, including annotating its key parameters. These parameters are abbreviated throughout the remainder of this article as initial solid chance (ISC) (ISC—the percentage chance that a cell is solid at the start of the simulation); birth limit (BL—the neighbor count for dead cells to become live); death limit (DL—the neighbor count for live cells to become dead).

This is one of the sample generators provided with Danesh. We also include a chunk-based 2-D level generator in the style of Spelunky, and several maze generation algorithms. For this article, we use the cellular automata system as a running example for consistency.

IV. Automated Analysis in Danesh

In [8], Smith and Whitehead propose ERA as a way to analyze and visually represent the distribution of a generator's output. In Danesh, we provide a way to automate this process, allowing users to examine generators through the lens of the metrics they supplied, and observe how the expressive range changes as they change the value of parameters.

When asked to perform an ERA, Danesh samples the generator in question many times—500 by default—and records values for all metrics currently defined by the user. The user can then request an ERA histogram for any two metrics, which is generated dynamically by Danesh from the gathered data. An example ERA from Danesh is shown in Fig. 4, with the metric selection visible at the bottom of the screen.

An ERA provides the user with an understanding of how the generative system behaves under the current set of parameters. Danesh can also perform a randomized ERA, or RERA, which provides the user with a wider picture of how the generative system behaves across its parameter space. It does this by performing an ERA with a much higher sample size (ten times larger by default) and for each sample, it randomizes the parameter settings between maximum and minimum values. The resulting histograms provide a sense of the distribution of outputs across all parameterizations. This can reveal surprising areas of the parameter space, as well as indicate areas of the metric space

```
[Generator]
public Tile[,] GenerateMap(){
//...

[Tunable(MinValue: 4, MaxValue: 8)]
public int mapWidth;

[Metric]
public static float Density(object _map){
//...

[Visualiser]
public Texture2D RenderMap(object _map){
//...
```

Fig. 3. Annotated generator, parameter, metric, and visualizer.

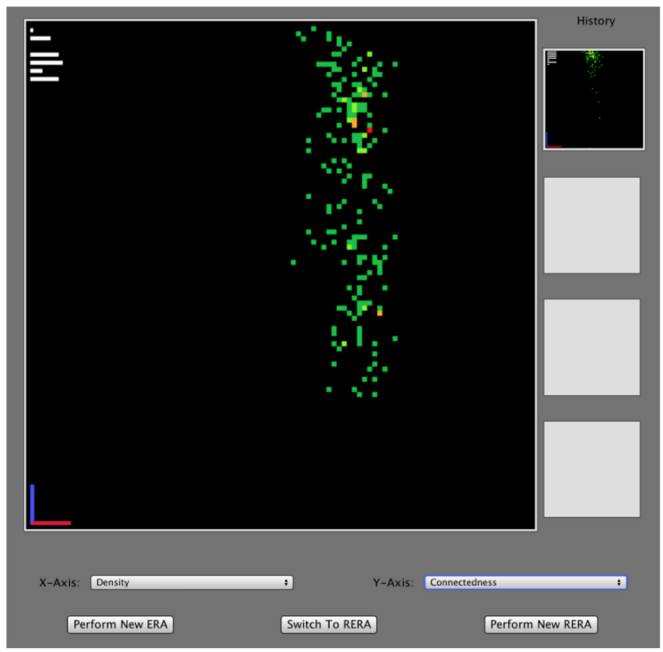

Fig. 4. ERA in Danesh.

that are unreachable. If an RERA shows that an area of 2-D metric space has no data points in, it means that across all random parameter settings there were no outputs with these metric values. This can suggest that regardless of any parameter tweaking, the generator cannot produce outputs with these metric values, which can be useful information for someone attempting to obtain a particular output from the generator.

Both ERA and RERA visualizations in Danesh are interactive. If the user hovers their cursor over any datapoint, they are shown an output from the sample that had these metric values. This is useful because it allows the user to make sense of distributions. Rather than simply knowing an outlier exists, the user can see the output in question and understand what makes it an outlier. In an RERA, the user can click on a point in the visualization to set the current generator to the set of parameters that produced this output. This makes RERAs a useful exploratory tool, as the user can hover over points with metric values they are interested in,

see what the outputs look like, and click to configure their generator to produce this without adjusting parameters manually. This is a good example of how Danesh helps shift the emphasis in interaction from being focused on inputs (editing parameters and code) to being focused on outputs (thinking about desired metrics and output styles).

V. Automated Parameterization

ERAs help the user visualize the current behavior of their generator, and RERAs help explore the range of possible outputs from a system. These are both useful tools for thinking about a procedural generator in different terms, and by making RERAs interactive they become useful exploratory tools for seeing different points in a generator's parameter space. An individual point in an RERA is often unrepresentative, however—after clicking on a point in the RERA, the user may discover the parameterization that produced that sample behaves unexpectedly, such as a very high variance in output, or an unexpected value in some other metric.

Some users have specific goals in mind for their generator, and can express these goals clearly in terms of the metric values they intend the output to have. These desired values, interpreted as the average generator output, are equivalent to the centroid of the ERA this desired generator configuration would have, since the centroid of an ERA describes the average metric value of a large sample of generative outputs. In this example, the user is trying to find a set of parameters that results in a generator whose ERA has a centroid closest to a desired set of metric values.

This is a difficult problem to solve manually. ERAs can show how a parameterization is located in 2-D metric space, but they cannot help the user know what changes to make to improve the result, because the impact of parameter alteration is often nonlinear. Exploring through an RERA is also possible, by clicking on points that are near to the desired point in metric space. However, this has the following two limitations: RERAs can only be shown in two dimensions in Danesh, and the user may have goals set in three or more metrics; and a sample in an RERA is not representative of that parameterization and, thus, is equally likely to be an outlier as it is to be near the centre.

To help overcome this, we have developed an automatic parameterization process, in which Danesh searches the parameter space in order to find a set of parameters whose ERA centroid is as close as possible to a set of metric values provided by the user. This process uses a hybrid of random search and hill climbing: Danesh spends 5 s randomly searching parameter settings, keeping the best results, and then uses hill climbing seeded with the best output found through random search. This process can be run for longer to find better parameterizations. The process primarily scores a parameterization by the difference between its centroid and the target metric scores, which means it does not take into account the variance of a generator. This means that this process can generate parameterizations with a very large distribution of outputs, or with almost no variance whatsoever. Incorporating a target variance into this process remains a point of future work—although simple in principle, communicating this to the user is a challenge. We describe this work on automated parameterization in greater depth, with additional experimentation and comparisons with other search strategies, in [17].

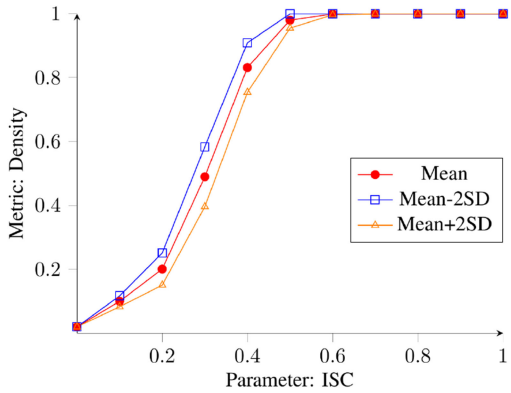

Fig. 5. Smoothness analysis, comparing the metric density against the parameter ISC for a cellular automata-based generator.

VI. Parameter Behavior Analysis

The parameters that Danesh presents to a user are derived from fields in the generator's codebase, annotated as we described earlier in this article. Their existence is largely dictated by the design and structure of the generative algorithm and the rest of the game's code, as well as being influenced by the specific programming style of the person who implemented the generator. This means that there is often not a clear connection between a parameter's value and the generated content, and this connection can be even harder to intuit if the user of the generator did not write the code.

To help mitigate these issues, we have developed new analytical techniques that aim to provide a better understanding of how a particular parameter behaves, and what the effect of changing it is likely to have on the generator's output. Our aim is twofold: to provide more complex, deep analytical techniques for the expert users who can make sense of them; and to augment the understanding of novice users with subtle additional information that does not require complex analysis. We have developed two new analytical techniques: smoothness analysis and codependence analysis. Each is useful on its own as a way of investigating a generator's behavior, but they have also enabled us to develop new kinds of interface features for Danesh.

A. Smoothness

Smoothness analysis visualizes how the value of a particular metric function m changes as the value of a parameter p varies between its minimum and maximum value. Fig. 5 shows an example smoothness analysis for one of the example generators supplied with Danesh, which generates cave-like maps using cellular automata. This example shows how the metric density changes as the value of the parameter ISC changes. For each value of ISC (sampled at regular intervals of 10% of the maximum value), we sample the generator 250 times and calculate the average metric value for the sample, as well as the standard deviation. We then plot this information on a graph, as shown in Fig. 5.

As can be seen in this example, changing the value of the parameter by the same interval can have very different impacts on the chosen metric. Moving from 0.2 to 0.3, a shift of 0.1, has a much smaller impact on the average density than moving from 0.3 to 0.4. Similarly, changing the value of the parameter past 0.5 has no effect on the metric at all. We argue that by default, in the absence of any other information about the system, a user would expect a parameter to exhibit perfectly linear smoothness—that is, a straight-line graph with gradient 1 passing through (0,0). Smoothness analyzes show how parameters deviate from this expectation, and specifically which parts of their range this deviation occurs in and to what degree.

B. Applications of Smoothness

Smoothness is another useful tool for generative systems developers, allowing the user to focus on the relationship between features of the inputs and outputs of a system. However, in addition to being useful in isolation, smoothness can be used by tools such as Danesh to automatically adjust and augment their interfaces. We use smoothness analysis to perform parameter smoothing, a process by which a parameter's original input slider is replaced with a more natural façade. Normally, a slider on Danesh's interface represents a linear range from the minimum and maximum value a parameter can take. The slider represents a value v in the range [0,1], which sets the value of the parameter p to the following value:

$$p = p_{\text{min}} + (p_{\text{max}} - p_{\text{min}}) \times v$$

where p_{min} and p_{max} represent the minimum and maximum values p can take, respectively. Parameter smoothing replaces this slider with a smoothed slider relative to a single metric m. First, the system performs a smoothness analysis of p against m. Let m_{min} and m_{max} be the minimum and maximum value recorded for the metric m during the smoothness analysis. Instead of the smoothed slider value v_s interpolating between the range of values p can take, it instead interpolates between the possible metric values. That is, we calculate the target metric value m_{tgt} as follows:

$$m_{\text{tgt}} = m_{\text{min}} + (m_{\text{max}} - m_{\text{min}}) \times v_s.$$

We then record the corresponding parameter value for m_{tgt} in the smoothness analysis (i.e., we find the p_{tgt} such that (p_{tgt}, m_{tgt}) lies on the smoothness graph). We set the parameter p to this value. This has the effect of setting the parameter value based on the expectation of linear smoothness: moving 10% along the slider always results in moving 10% along the range of metric outputs, the user is simply unaware of the actual impact on the parameter being edited. Figs. 6 and 7 show outputs from the cellular automata cave generator mentioned earlier. In Fig. 6, the outputs are from the unsmoothed parameter slider, setting the parameter to one-quarter intervals. Note how the caves are very fragmented at 50% slider value, and completely solid by 75%. In Fig. 7, we see the same slider values but for a smoothed slider. The 50% point on the slider now represents a 50% point between maximum and minimum density, and at 75% there is still a lot of open space. The useful range of parameter values has been extended to take up more of the slider's range.

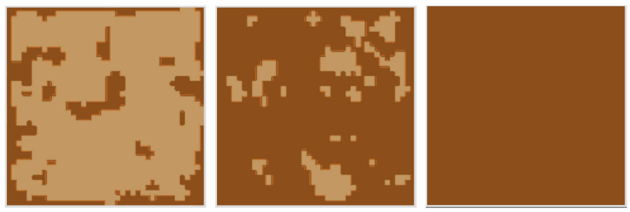

Fig. 6. Results from three intervals across the range of an unsmoothed parameter slider, e.g., 25% represents a parameter value 25% between minimum and maximum value. (a) 25% of range. (b) 50% of range. (c) 75% of range.

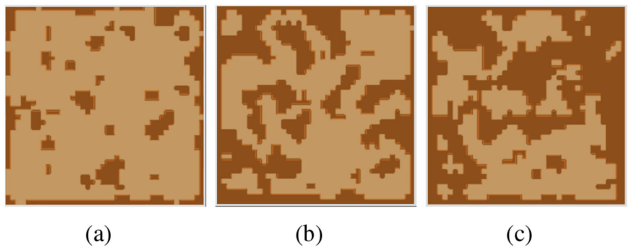

Fig. 7. Results from three intervals across the range of the smoothed version of the slider shown in Fig. 6. (a) 25% of range. (b) 50% of range. (c) 75% of range.

Currently, parameter smoothing can only be applied to parameters whose smoothness is monotonically increasing or decreasing. In [18], we suggest that parameter smoothing may be possible on nonmonotonically smooth parameters by partitioning the smoothness curve into segments, each of which are either monotonically increasing or decreasing, and smoothing each segment individually. We intend to pursue this in the next major version of Danesh.

C. Codependence

Codependence is an extended form of smoothness analysis that visualizes the relationship between two parameters p_1 and p_2 with respect to a metric m. Smoothness analysis attempts to resolve confusion arising out of unexpected behavior when changing a parameter; whereas codependence analysis attempts to solve another problem, in which a change to one parameter causes a change in behavior in another parameter. To perform a codependency analysis, we perform successive smoothness analyzes of p_1 with respect to m. For each smoothness analysis, we vary the value of p_2 between its minimum and maximum values at regular intervals, similar to how we vary the value of p_1 in a regular smoothness analysis. The result is a series of smoothness graphs, which can be plotted together as a 3-D surface, as shown in Fig. 8.

If two parameters are independent of one another, we expect the codependence surface to exhibit symmetry along the axis of p_2. However, if the parameters are codependent on one another, we will see a lack of symmetry. In Fig. 8, we see that the shape of the surface changes along both axes. This means that as we change the value of BL, the smoothness of DL changes, and vice versa. This is not necessarily an effect that can be avoided. In this example, the two parameters are intrinsically linked as part of the

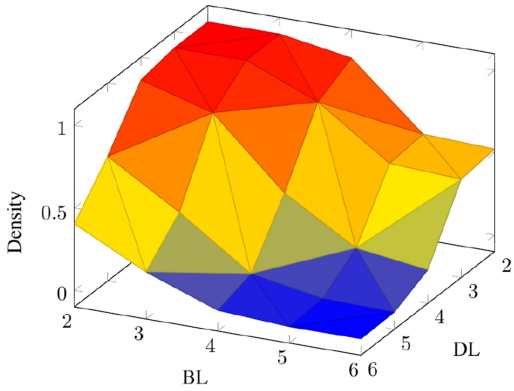

Fig. 8. Codependence analysis, comparing the metric density against the parameters BL and DL for a cellular automata-based generator.

Fig. 9. Parameter sliders, highlighted with colors indicating the effect of the change just made by the user.

algorithm. Codependence analysis allows us to see exactly how this link manifests, however, and can also help us understand the nature of the linking. This can help generative systems designers understand unusual relationships in their systems. For example, Fig. 8 has diagonal symmetry, implying here that the density metric is related to the sum of the two parameters.

D. Applications of Codependence

In addition to being a useful form of analysis on its own, codependence can also help us augment the tools we design, similar to automatic parameter smoothing described earlier in this section. In Danesh, we have prototyped automatic codependence highlighting for parameters to warn the user of the anticipated impact of changes they make. Fig. 9 shows an example of automatic codependence highlighting in Danesh's interface.

As the user makes a change to a parameter, we measure the difference between the smoothness of each parameter before and after the change (i.e., we sample two slices from the codependency analysis using the old and new parameter values). We then recolour the name of each parameter based on the distance between the two slices—redder colours indicating a larger distance, and thus, a larger expected shift in behavior. This gives users a way of anticipating how the behavior of their system changes as they interact with it, without exposing them to the complexity of the raw analysis.

For more information on smoothness and codependency analysis, with additional examples and discussion, see [18].

VII. Automated Parameter Discovery

Earlier we described the initial setup of Danesh, which normally involves annotating fields in the game's codebase so that they are recognised as parameters. These parameters are then used by Danesh's users, as well as being important for some of Danesh's more complex features such as automatic parameterization.

From a user perspective, having to manually identify parameters poses a few potential problems. Recall that one of the aims of this tool is to provide a powerful and intuitive way to interact with procedural generators for nonprogrammers, or for less experienced users who are working with code written by a third party (e.g., open-source code found online, or a tool written by a team member). In this instance, the user may find it hard or impossible to know what fields need annotating. At the same time, even experienced users who are working with their own code may not be aware of all of the fields that impact the generator's output, and might benefit from a deeper analysis of the generator.

To solve these problem, Danesh can perform an automated parameter discovery process to identify fields, which could be used as parameters, and then integrate them into the tool without needing manual annotation. This not only lowers the barrier to entry for new users, but it can also provide surprising discoveries and helps foster a deeper understanding of the system for expert users.

1) Parameter Assessment: Danesh performs parameter discovery by searching through the entire codebase using C#'s metaprogramming features to look for any accessible field (in its current implementation, it also ignores visibility modifiers, such as `private`). We currently only consider fields, which have a numeric type (e.g., `float`, `int`) or a boolean type. When a field is discovered through this search, Danesh performs tests to assess whether it has an impact on the generator.

Before performing any assessment, we record a baseline for the current behavior of the generator. We perform a similar process to that of an ERA: we sample the generator 100 times, and for each metric function m, we calculate the average value, $\overline{m}$, and standard deviation, m^{ϕ}, for each defined Metric function. We also record the time taken to calculate this sample, which we use later.

We then attempt to discover plausible upper and lower limits for the parameter, by slowly growing limits out from a default starting point. An interval value, δ, is initially defined based on the field's type: 1 for integer types, 0.1 for floating point types. We then create two new variables v_u and v_l representing the upper and lower bound of the field, respectively, and initialize them to the original parameter value.

We then repeat the following process: add δ to v_u and set the field's value to the new value; calculate a new set of sample outputs and calculate $\overline{m}$ and m^{ϕ} for the generator's new configuration; record these values; increase the value of δ. We repeat this process for a fixed number of iterations (currently 100 iterations by default) or until one of three early stopping conditions are met. The first is any catchable exception is thrown, this suggests the parameter has been set outside of expected bounds. The second is that the time taken to generate a sample exceeds ten times the original generation baseline, this suggests it has exceeded the reasonable range intended for the system. Finally, if the change in $\overline{m}$ between this value for v_u and the previous value is less

than the standard deviation for this sample, for all metrics, we consider it to have not meaningfully changed, and terminate. That is, if

$$|\overline{m}^{\text{new}} - \overline{m}| \leq m^{\phi}.$$

This process is repeated for v_l as well, subtracting the δ each iteration rather than adding it. Each iteration, the value of δ is increased by $10 \times (i+1)$ where i is the current number of passed iterations. Once both the upper and lower bound searches have terminated, this gives us a maximum range to set the parameters within. However, in the event of early termination due to crashes or time penalties, the last values for the bounds may not be valid (i.e., they may result in crashes or excessive execution times). Therefore, we perform another round of search to find a good bounds value in the range between the last and penultimate values for each bound. Let v_b be the bound value being finalised, and v_p be the penultimate value of the bound that is the bound in the last-but-one iteration. We repeatedly set $v_b = \frac{v_b + v_p}{2}$ and then retest the new value of v_b. If it causes a crash or exceeds the time limit, we repeat the process until it passes both tests.

This provides us with a good estimation of the absolute limits for the parameter that both do not negatively affect the procedural generator, while also ensuring they have a meaningful impact on the procedural generator. We then present the user with a final confirmation prompt. This allows the user to rename the parameter (by default, we suggest the field's name) and they can also edit the upper and lower bounds if they disagree with the discovered values. Once they confirm, the parameter appears in the parameter list, as seen in Fig. 1.

2) Preliminary Experimentation: In our tests, automatic parameter discovery has managed to rediscover all of the manually annotated parameters in our example generators. It made several other suggestions, which were not manually tagged but made sense to modify (such as the dimensions of the map for a maze generator), and in one case found a parameter that surprised us. The surprising parameter was attached to a world map generator based on the diamond-square algorithm; Danesh suggested controlling a parameter, which governs the rate at which the algorithm executes and terminates. We had not considered using this to customize the generator, but by changing it to lower or higher values it simulates a "camera zoom" effect on the map, effectively changing the resolution of the underlying random noise. This shows how even in simple example generators this technique can find unusual control affordances.

In terms of bounds estimation, Danesh is able to find safe boundary values for parameters with a good confidence—in our experiments on three generators with over a dozen expert-configured parameter bounds, none of Danesh's bounds estimates resulted in crashes, slow processing or other undesirable outputs. There are two weaknesses to the process, which stand out immediately, however. The first is that Danesh cannot identify common-sense boundaries for parameters, such as those with an implicit constraint of being greater than zero. In many cases setting these parameters to a negative value does not actually cause a crash, but is something that could never actually happen in practice, and thus, Danesh's suggestions are less useful. The second is that because we do not know how many parameter candidates Danesh might find, we have tuned the search to not spend too long on each bounds estimation, which can result in Danesh not refining the bounds as much in the second phase.

Despite these limitations, this feature works well for its intended purpose, which is to provide the user with a set of safe initial values while they investigate the affordances of a newly discovered parameter. An expert user will likely be able to quickly set better values, while a less experienced user is guaranteed that this range of values, although possibly esoteric, will not crash the system and will provide varied output.

One limitation of the search process is that it is unaware of side effects in the codebase. When Danesh tests a field to see if it can be used as a parameter, it records the original value and sets it back to this value after testing. However, during the testing process, it can potentially run the generator in a configuration that the user did not expect, which can cause values elsewhere within the generator to be updated. Consider a simple example where there is a boolean which, if set to true, causes some user data to be reset when the generator is run. While this boolean can be set back to its original value, the side effect of resetting the user data cannot be anticipated or easily detected.

While most of these changes would be reset when unity is closed, they persist while Danesh is running and so will affect future interactions with the tool until the program is shut down and relaunched. These changes may be hard to detect, especially for users unfamiliar with the codebase, and often affect fields which are not surfaced by unity's user interface, which makes them hard to reset without a full relaunch of the software. One of the reasons for this arises from how unity handles default values for objects. If an object exists in a game scene in unity, public fields in its scripts will be visible in the tool and can have their default values changed. This overrides any default values set in code. However, Danesh's parameter discover process can change the values of nonpublic fields. Despite not being displayed in unity's interface, these changes nevertheless exhibit the same kind of overriding behavior, which we believe is unity detecting a change in the value and persisting it even though the field is not displayed or editable in the interface.

As a result, this feature has remained labeled as experimental. While we believe the fundamental technique is useful and effective, it is not entirely reliable or safe to use in the current environment. In the future, we are considering a version of Danesh that is disconnected from unity, which would make it easier to control, which changes persist, as well as making it easier to detect side effects during execution. We also plan to test this feature on larger generators as part of more established codebases, with many more parameters to test and evaluate.

VIII. Lessons Learned

A. Platform Development

A crucial decision made early in Danesh's development was whether the software should be developed a standalone tool, or as part of an existing platform (and if so, which one). An important aim of ours was to support people across a wide

spectrum of abilities and, therefore, embedding the tool in an existing platform that was already well-known to students and hobbyists made sense. We chose unity over its competitors (in particular, unreal has grown in popularity over the last few years) partly because it has a reputation for being used by people new to game development, as well as for adjacent domains such as digital art projects, while unreal and cryengine are seen as tools more exclusively targeting professional use.

There are several drawbacks to embedding the tool in an existing platform. Despite its popularity, unity is not used by every game developer, and thus, there are many people unable to use Danesh in its current form. Developing the tool as a standalone application would have made it more universal. There are also stigmas associated with unity (both from a technical performance standpoint, and a consumer perception of engine quality), although this is not a huge concern for us as we are mostly targeting existing unity users.

Using an existing platform had many advantages. Unity's built-in user interface API is relatively simple to use, and allowed us to quickly prototype and iterate on Danesh's interface. Although the interface is not particularly nice to look at, it is functional and smoothly integrates with unity's existing toolset. Developing a plugin for unity also allowed us to distribute the plugin on unity's Asset Store, which extended our reach far beyond simply advertising an open-source project. For an academic project, however, there are downsides to the asset store: while we initially received favourable reviews for the tool, a new version of unity was released, which caused Danesh to be incompatible. Despite the compatible versions being clearly marked on the Asset Store page, we nevertheless received complaints and negative reviews for not updating the plugin fast enough. Without a dedicated developer, it is hard to guarantee support for a tool like this, but this is an expectation many Asset Store users had of anything listed online.

B. Generality and Usability

One of Danesh's most important design tenets is that it is agnostic to the type of content being generated, meaning it can be adapted to work with generators working in any kind of domain, as well as generative systems outside of videogames. This was important to us because of the breadth of application areas for generative software, and the biased focus on certain types of content in procedural generation research. In order for this generality to be possible, Danesh requires the user to define parts of the system, which require specific domain knowledge. Chief among these are the visualization code that defines how a piece of content is displayed on the screen, and the metrics that measure features of a piece of content, which Danesh uses in almost all of its analytical processes.

This design choice provided Danesh with a huge amount of flexibility and power, not only in terms of supporting any kind of generative software, but also the flexibility to mix and match this code, switching between visualizers for a generator, or applying the same metrics to two different versions of a level generator. This generality comes at the expense of usability, however, because a novice user may not know how to write a visualizer, or may not know what kind of features they are most interested in as metrics. This was a significant drawback for Danesh that limited its usefulness to nonprogrammers. In particular, we viewed metric definition as part of the initial setup and configuration process of using Danesh but for many users, both expert and novice, this is actually an important part of the process of understanding the procedural generator and the designer's goals. In the future, we wish to do more work developing this aspect of the tool, and making this part of the process better guided.

IX. Future Directions for Generative Analysis

A. Composite Procedural Content Generation

If a user is unhappy with the way their generator produces content, the most common course of action taken is to adjust the values of the parameters to change the distribution of the generated content. This is a time-consuming task, and one which Danesh was designed to alleviate, by developing better visualization and feedback for editing, as well as by introducing automated techniques for searching the parameter space.

This approach to designing procedural content generators has drawbacks. It assumes the existence of a single parameterization that satisfies all of the designer's criteria, which may not exist, particularly if it involves offering a large distribution of content types with conflicting features (for example, a cave generator that sometimes generates winding mazes, and other times generates large arenas). Even with automatic search, it may not be possible to satisfy designer goals.

It is also a very limited way of thinking about content generation. Commonly in games, a content generation task is solved by implementing a single procedural content generator and always sampling from it. Games occasionally break from this pattern—*No Man's Sky*, for instance, provides different template corpora to its generator based on the planet type, so a very hostile creature will draw body components from a different set of 3-D models than a docile creature. This provides higher level variety—content is sampled not just from a single distribution, but many distributions defined by the selected corpus of source material.

We propose a generalization of this approach, which we call composite procedural content generation (CPCG), whereby a procedural generator is modeled not as a single algorithm but as a weighted sampling of many different generative algorithms producing the same kind of content, which may be parameterizations of the same generator, or different algorithms altogether. Note this is different from orchestrating generators producing different kinds of content [19]. Instead, our aim is synthesise a new generative space as a union of several existing generative subspaces.

A simple CPCG system would sample from several different parameterizations of the same generator, with different probabilities assigned to each paramaterization. Danesh would be able to show the ERA both of individual parameterizations, as well as an ERA of the whole CPCG system, allowing the user to inspect individual generative systems as well as the distribution of the higher level meta-generator.

B. Interactive Tool Configuration

A recurring piece of feedback for Danesh is the difficulty of initially configuring the system. We have made many improvements in this area, such as automatic parameter discovery and automated metric searches. The definition of metrics, however, is a particularly important part of using the tool, and the quality of metrics depends on the user. Even in expert use cases, users may not know what they are looking for, and the metrics they write are limited by the properties of their content they are currently considering.

We believe that an important aspect of future generative assistance tools is in being able to help the user explore their own needs and interested as part of using and configuring the tool. This could mean, for example, being able to interactively define metrics by identifying positive and negative examples of a particular property and using machine learning approaches to try and infer possible models for metrics from this. The use of curious agents to explore the possibility space and identify styles, clusters, or interesting features will also help users come to terms with the vast spaces and combinatorial complexity of the generators they are working with.

Even for tools that are more specialised than Danesh, perhaps focusing on a specific generative use-case such as Ropossum, this kind of research is increasingly important. As we described in Section I when quoting Grant Duncan, users are keen for more control and understanding of generative systems. Tools like this must be able to teach their users as they work, because each generative system is in itself a new learning experience, with its own unique foibles and eccentricities. As we stated earlier in this article, one of our goals with Danesh was to shift the user's focus from input parameters to thinking about outputs and the function of the generator within a game or system. This can be achieved through better visualization and analysis, but also needs better ways for the user to express intent and interest, too.

X. Conclusion

In this article, we describe the development of Danesh, a tool for analyzing procedural generators that plugs into the popular unity game development environment. We showed how Danesh was designed to be as general as possible, allowing it to be connected to a game's codebase directly, with customized visualizations, and user-written evaluatory metrics. We described how we use this foundation to provide rich analytical capabiltiies, including automated ERA, interactive generative space search, interaction smoothing, and parameter discovery. These features show off a range of techniques for working with generative systems, from fully automated to fully user-driven. All of these approaches yield interesting results and lend themselves toward different kinds of aims when using the software.

We also used this article as an opportunity to reflect on the development of the project for the first time. We discussed the tradeoffs made in trying to build a tool that was as general as possible while still being useful, and how our decision to integrate with unity worked.

Generative systems require a new and unusual way of thinking about data, content, and processes in order to get the most out of them. Yet we lack good ways to communicate and teach these ideas and modes of thinking, which means that ideas like procedural generation often gain a reputation for being obtuse, hard to learn and resistant to control. Many things can be done to overcome these issues, but chief among them is to build tools that help people access and explore these ideas in a structured way, and without relying purely on reading program code.

Acknowledgment

The authors would like to thank J. Van Hove for his support of Danesh, and the reviewers for their insightful comments.

References

[1] *Interactive Data Visualisation*, "Speedtree," 2000. [Online]. Available: http://www.speedtree.com
[2] P. Weir, "The sound of No Man's Sky," 2017. [Online]. Available: https://tinyurl.com/nmssound
[3] Inkle, "Ink (language)," 2021. [Online]. Available: https://www.inklestudios.com/ink/
[4] G. Duncan, "No Man's Sky: How i learned to love procedural art," 2015. [Online]. Available: https://tinyurl.com/nmsartgdc
[5] D. Saas, "No Man's Sky evokes wonder through math," 2016. [Online]. Available: https://tinyurl.com/nmspress
[6] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: A mixed-initiative level design tool," in *Proc. 5th Int. Conf. Found. Digit. Games*, 2010, pp. 209–216.
[7] A. Liapis, G. N. Yannakakis, and J. Togelius, "Sentient sketchbook: Computer-aided game level authoring," in *Proc. 8th Conf. Found. Digit. Games*, pp. 213–220, 2013.
[8] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proc. FDG Workshop Procedural Content Gener. Games*, pp. 1–72010.
[9] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha, "Launchpad: A rhythm-based level generator for 2-D platformers," *IEEE Trans. Comput. Intell. AI Games*, vol. 3, no. 1, pp. 1–16, Mar. 2011.
[10] A. Canossa and G. Smith, "Towards a procedural evaluation technique: Metrics for level design," in *Proc. 10th Int. Conf. Found. Digit. Games*, pp. 69–77, 2015.
[11] J. R. H. Mariño, W. M. P. Reis, and L. H. S. Lelis, "An empirical evaluation of evaluation metrics of procedurally generated Mario levels," in *Proc. 11th AAAI Conf. Artif. Intell. Interactive Digit. Entertainment*, pp. 41–49, 2015.
[12] A. Summerville, "Expanding expressive range: Evaluation methodologies for procedural content generation," in *Proc. 14th AAAI Conf. Artif. Intell. Interactive Digit. Entertainment*, pp. 420–428, 2018.
[13] A. Summerville *et al.*, "Procedural content generation via machine learning (PCGML)," *IEEE Trans. Games*, vol. 10, no. 3, pp. 257–270, Sep. 2018.
[14] J. Zhu, A. Liapis, S. Risi, R. Bidarra, and G. M. Youngblood, "Explainable AI for designers: A human-centered perspective on mixed-initiative co-creation," in *Proc. IEEE Conf. Comput. Intell. Games*, pp. 1–8, 2018.
[15] J. Xie, C. M. Myers, and J. Zhu, "Interactive visualizer to facilitate game designers in understanding machine learning," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, pp. 1–6, 2019.
[16] M. Cook, "Generate random cave levels using cellular automata," 2013. [Online]. Available: https://tinyurl.com/cook-caves
[17] M. Cook, J. Gow, and S. Colton, "Towards the automatic optimisation of procedural content generators," in *Proc. IEEE Conf. Comput. Intell. Games*, pp. 1–8, 2016.
[18] M. Cook, J. Gow, G. Smith, and S. Colton, "General analytical techniques for parameter-based procedural content generators," in *Proc. 1st IEEE Conf. Games*, pp. 33–39, 2019.
[19] A. Liapis, G. N. Yannakakis, M. J. Nelson, M. Preuss, and R. Bidarra, "Orchestrating game generation," *IEEE Trans. Games*, vol. 11, no. 1, pp. 48–68, Mar. 2019.