Towards The Automatic Optimisation Of Procedural Content Generators

Michael Cook
Games Academy
Falmouth University
mike@gamesbyangelina.org

Jeremy Gow
Computational Creativity Group
Goldsmiths, University of London
j.gow@gold.ac.uk

Simon Colton
Games Academy
Falmouth University
simon.colton@falmouth.ac.uk

Abstract—**Procedural generation is important to modern game development as both an artistic implement and an engineering tool. However, developing procedural generators and understanding how they work are both difficult tasks, and even more so for novice developers. In this paper we describe Danesh, a tool to help in analysing, changing and exploring procedural content generators. In particular, we describe several features in Danesh which help a user optimise their procedural generator towards a certain kind of output by automatically changing parameters and evaluating the effect it has on the generator. We compare different approaches to these tasks and describe our future intentions for Danesh's automated features.**

I. Introduction

Procedural content generation is an important part of modern game development and a well-known concept among gamers and critics [1], useful both as a tool for solving problems [2] and a paintbrush for expressing ideas [3], and often employed as both at once [4]. The ability to generate game content automatically opens up the potential for new kinds of game to be made possible [5], as well as easing the development of games by allowing abundance, serendipity and surprise to be added to a game in very simple ways [6].

In our experience, developing procedural generators is hard. Unlike many other common concepts in game development like physics engines [7] or particle systems [8], most modern tools for making games do not come with built-in support for content generation. Where they do offer such support, it is usually in the form of fixed-purpose generators of (usually decorative visual) content, such as Unity's tree generator [9]. In addition to this, a user wishing to learn more about procedural generation is mostly limited to tutorials about generating highly specific content through one method (such as Unity's cave generator tutorial [10]), which usually focus on the process of implementation rather than understanding and customising a generator after the fact. As an example, the tutorial in [11] ends by telling the reader: 'the final step is down to you: you must iterate over what you learned to create more procedurally generated content for endless replayability'.

Understanding procedural generators can also be difficult. Unlike traditional content creation, the user cannot see the entirety of what they are creating all at once. Instead, they typically view an example output from the generator (perhaps multiple examples in quick succession) to assess the approximate type of output the generator might produce. In a survey we conducted of 53 game developers, 85% of them stated that their primary method of testing a generator involved changing parameter values and repeatedly viewing the output. There are many problems with such an approach: it doesn't capture the variance or distribution of the output; it can't detect outliers or anomalous results; it is extremely hard to do when parameters have nonlinear relationships with the output or interact with other parameters; and it doesn't provide feedback to the user as to whether their goal is even achievable.

In order to help new practitioners learn about procedural content generation as a skill and an artform, we need to build tools that focus on domain-independent analytical techniques, that provide as much help and feedback as possible in the process of learning what a generator does and understanding how to change it. There is also a need for such tools among experts too – better development tools could streamline the process of working with procedural generators and help make them more accessible to artists and designers [12]. This would make their use in modern game development easier, but also promote experimentation among expert communities [13].

In this paper we describe Danesh, an open-source tool for helping designers and developers of all experience levels explore, explain and experiment with procedural content generators. We focus here on describing our techniques for automatically optimising and analysing the generators, including identifying useful parameters for the generator to tweak and automatically configuring the generator towards producing a certain kind of output. In addition to being useful for both novice PCG developers and experts, we hope that by developing Danesh as an open-source tool we can promote crossover between procedural generation researchers, and provide an open platform for implementing analytical ideas about generative software.

The remainder of the paper is organised as follows: in section II we discuss existing work at the intersection of procedural generation and design tools; in section III we introduce Danesh and describe some of its features, including a suite of analysis tools; in section IV we describe the ways in which Danesh can automate some of its processes, and describe experimentation done to assess the best techniques; in section V we discuss the strengths and limitations of Danesh in its current form, and describe our intentions for future work on the system; finally, in section VI we summarise the paper's results.

II. Background

In [14] Smith and Whitehead describe Tanagra, a tool for generating levels for platforming games using a rhythm-based system for labelling and slotting content together. In this paper they introduce the notion of *expressive range*, a way of evaluating a generator based on the qualities of its output. Hundreds of pieces of content from the generator are produced and then evaluated according to two metrics (in the paper these are the linearity of a level and leniency of its difficulty structure). The values are then plotted on a histogram, with bright colours or larger points indicating that more pieces of content fall in a particular area. This provides a neat visual summary of the current state of the generator, as the user can see information such as the spread of the generator's output (whether the cluster of points is large or small); the correlation between axis metrics (whether the points appear to correlate along a particular line); the quantity and nature of outlying points; and areas of the metric space which the generator does not currently cover. We have implemented expressive range in Danesh and this enables the user to generate histograms based on those described in [14].

Besides Tanagra, other attempts have been made to produce tools for generating content, typically tailored to one specific kind of game content such as maps or levels. Ropossum [15] uses a physics simulation to verify levels for the game Cut The Rope, and can either generate levels from scratch or co-create with a human user who has produced a partial level design. The Sentient Sketchbook [16] is a slightly more general tool aimed at developing maps – users can sketch a map at a low resolution and then upscale the map to a more detailed version automatically using the tool. Danesh is related to this work primarily because all of these tools are concerned with generative software – however, Danesh is a general analytical and developmental layer intended to be plugged into an existing generator, whereas the systems mentioned above are all generators themselves, trying to help a user solve a particular type of content generation problem.

In [17] the authors describe a procedural procedural level generator generator. This represents another attempt at higher-level descriptions of generators, by describing the properties of a system which then goes on to generate procedural generators. The work describes the notion of 'inner' and 'outer' generators, where the inner generation process is parameterised by the outer level, which the user interacts with. Similar to the tools we have mentioned, this is a bespoke tool aimed at a particular game domain, although the techniques are quite broadly applicable. One of the most interesting features of this work is how playful and divergent the system is – interacting with the generator generator is an enjoyable experience, and can produce unusual and unexpected results.

In [18] the authors present procedural generation from the perspective of those who use it, and highlight the different metaphors practitioners use when discussing it. One of the motivations for the work is to explore the possibility of a 'shared language' with which to talk about procedural content generation across different communities and areas of expertise. The four metaphors proposed are *Tool* (something which acts as an extension of its user to achieve a goal); *Material* (something that can be shaped or manipulated into a particular form); *Designer* (something that solves a design problem independently or collaboratively); and *Expert* (something that holds specific knowledge about a domain and can interpret data based on this knowledge).

The work presented in [18] not only reinforces how widely-used procedural generation is, by people of diverse backgrounds and experience, but also how important it is to provide different ways of engaging with this technology, and to assist people in getting the most out of these ideas by providing different ways to think about and explore them. We believe the work on Danesh outlined in this paper continues some of these ideas by trying to provide new tools to help people better understand procedural generation.

III. The Danesh Tool

Danesh is a Unity plugin designed to help developers to analyse and improve generative software. It is being developed with the intention of making it as general as possible – it is currently not designed to evaluate a particular kind of content generator, and instead relies on modular subsystems to evaluate and visualise generated content while being agnostic to what exactly is being generated. This does not mean that Danesh is able to work with any kind of generator – there are limitations to the current version of the software, which we discuss later. However, it can handle a wide range of generator types and provides a suite of useful tools for analysing and improving them, which we briefly describe in this section.

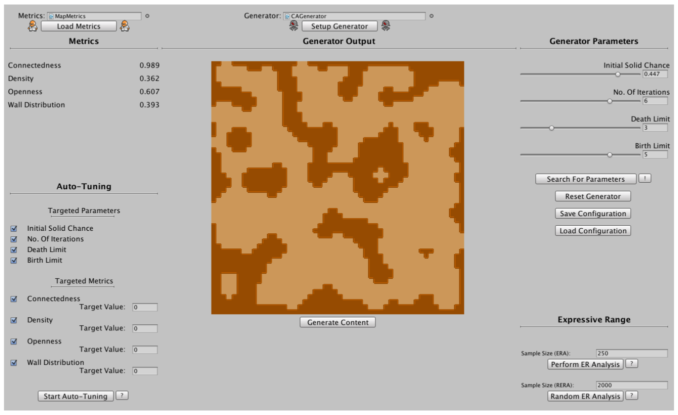
Danesh is an open-source C# project and can be downloaded from GitHub[1]. We are writing tutorials for new features of the tool as they are developed – these, as well as further information about Danesh, can be found on the tool's website[2].

A. The Cellular Automata Cave Generator

Throughout the remainder of this paper we will be using a cave generator based on cellular automata [20] whenever an example generator is needed for explaining a feature of Danesh. The implementation of the generator is based on [19]. The generator produces two-dimensional grids of solid and empty tiles which describe a top-down view of a cave-like structure. The implementation of the generator in Danesh has four parameters: the *initial chance* a tile will be randomly assigned as solid when the algorithm begins; the number of *iterations* the algorithm is run for; and two numbers that define the conditions for a tile changing from solid to empty, or empty to solid (called the *birth* and *death limits*).

Danesh uses what we call *metric functions* to measure features of a generator's output. These metric functions must be provided by the user before starting to use the tool. They are written as methods which take a piece of generated content

[1]http://github.com/gamesbyangelina/danesh
[2]http://www.danesh.procjam.com



Metrics: MapMetrics
Generator: CAGenerator

Connectedness	0.989
Density	0.362
Openness	0.607
Wall Distribution	0.393

Targeted Parameters: ✓ Initial Solid Chance, ✓ No. Of Iterations, ✓ Death Limit, ✓ Birth Limit

Targeted Metrics: ✓ Connectedness, Target Value: 0; ✓ Density, Target Value: 0; ✓ Openness, Target Value: 0; ✓ Wall Distribution, Target Value: 0

Initial Solid Chance: 0.447
No. Of Iterations: 6
Death Limit: 3
Birth Limit: 5

Sample Size (ERA): 250
Sample Size (RERA): 2000

Fig. 1: A screenshot of Danesh working with a generator.

and return a number in the interval $[0,1]$. We have written five metrics for the example cave generator:

- **Connectedness**: Measures the largest contiguous area of empty tiles, returns its area as a proportion of all empty tiles in the cave. (i.e. If the cave is one single contiguous area, returns 1.0).
- **Density**: Calculates the proportion of solid tiles.
- **Openness**: Calculates how many empty tiles are not adjacent to any solid tiles, as a proportion of all empty tiles.
- **Jaggedness**: Calculates how many empty tiles are 'jagged' (a tile is jagged if it is adjacent to two solid tiles on opposite sides), as a proportion of all empty tiles.
- **Wall Distribution**: Calculates how many empty tiles are adjacent to one or more solid tiles, as a proportion of all tiles in the cave.

The following sections will explain the use of these metrics and parameters in Danesh.

B. Loading, Viewing, Changing

Before loading in a generator, the user tags sections of their code with custom C# Attributes that mark out methods used for generating content and for visualising that content. Danesh can currently visualise content either as text, or as a 2D image. The user is relied upon to write the visualisation function for their content, although we provide a small suite of utility methods to help them do this easily. In order to load a generator, three things must be tagged with attributes: a `Generator` method which Danesh can call to generate a new piece of content, a `Visualiser` method which Danesh can call to display a piece of generated content on-screen, and any fields of the generator that the user wishes to change using Danesh's interface. When tagging a field, the user must also provide a simplified name to display in the tool, and minimum and maximum values for the field (where applicable).

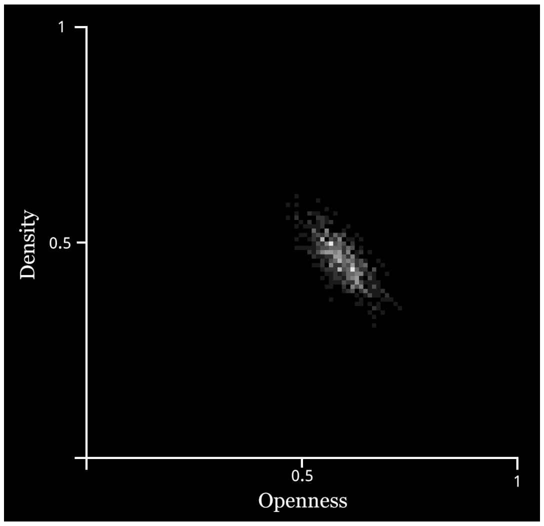
Figure 1 shows a screenshot of the main interface after loading a generator. The display is split into three columns. In the central column is a piece of content that has been generated by Danesh and displayed using the visualiser. In the right-hand column is a series of sliders, each one referring to a field in the generator that was tagged by the user. The slider is limited by the minimum and maximum values set by the user. Changing the slider values directly adjusts the underlying generator, and the user can then click a button to generate and display a new piece of content using the changed parameters.

C. Expressive Range Analysis

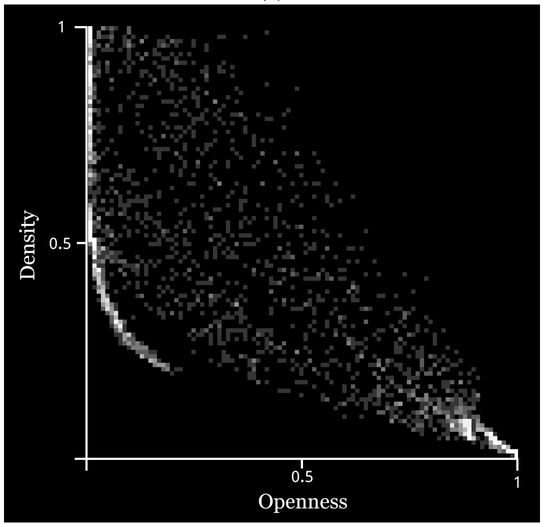
In the bottom-right corner of the interface in Figure 1 is a group of controls for the expressive range analysis functions of Danesh. Expressive range analysis (ERA) is a method of analysing a procedural generator according to the qualities of its output rather than those of its parameters. Suppose we have a generator with a set of fields and a zero-argument generation method `GenerateContent`. Additionally, suppose we have two metric functions `Metric1` and `Metric2` which take an output from the generation method and return a number in the interval $[0,1]$. To produce an expressive range histogram, we repeatedly sample from the generator and calculate both metric values for each sampled output. We record each result by multipling it by 100, flooring the result, and using the resulting integer to index a 100-element array, incrementing the indexed value. For example, if `Metric1` returns a value 0.87995, we calculate $\lfloor(0.87995 * 100)\rfloor$, which gives us 87, and then increment the value in the 87th element of the `Metric1` array.

To produce the visual ERA output we plot the data on a histogram with the two axes referring to `Metric1` and `Metric2`. If there are no samples at a particular co-ordinate, no colour is plotted. The higher the number of samples recorded at a point, the more intense the colour is plotted. This representation is derived from Smith and Whitehead's original expressive range histograms in [14]. Figure 2a shows an expressive range analysis of the cellular automata generator, as seen in Figure 1. The x-axis records *Openness* – the proportion of tiles in a level which have no solid tiles adjacent to them – while the y-axis records *Density* – the proportion of tiles in a level which are solid (regardless of what they are adjacent to). From the ERA in Figure 2a, we can see that the generator produces levels which contain about 50% solid tiles, and of the open tiles, just over 50% are not touching solid tiles. This might mean that the space in the dungeon is more open rooms than narrow corridors. The user can get more information about a data point in the ERA by hovering their mouse over it to view an example piece of content.

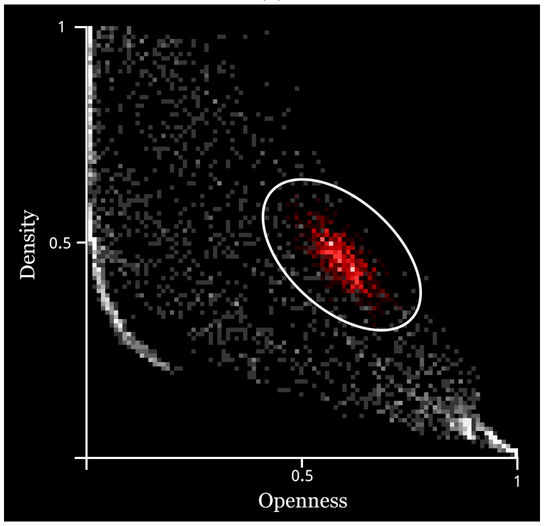
An ERA provides a visualisation of a single generator configuration, because it samples the generator with a set of fixed input parameters. This is useful when considering a particular configuration in detail, however it is not especially useful when the user wants to consider what might be possible with other configurations of the system. To this end, Danesh also provides a *randomised* expressive range analysis, or RERA. A RERA is performed very similarly to a standard ERA, except that each time Danesh samples the generator, the parameters are randomised within the minimum and maximum values set when the generator was loaded. The resulting histogram shows a broader picture of the generator's potential generative space. Figure 2b shows a randomised expressive range analysis of the



(a)



(b)



(c)

Fig. 2: Top: An example of an expressive range histogram. Middle: An example of a random ER histogram. Bottom: The ERA from 2a overlaid in red on the RERA from 2b. We have circled the area for readers of a B&W version of the paper.

same procedural generator which produced the ERA in Figure 2a. Here we can see a much wider set of potential outputs from the generator, as well as seeing which areas are more dense with content than others – or even which parts of the metric space the generator does not appear to cover at all.

Figure 2c shows the standard ERA overlaid in red on top of the RERA. From this image, we can see the current configuration of the generator in the wider context of its potential. We can see, for example, that we can increase the openness or the density of the generated content, but not both (the top-right of the RERA is completely dark, indicating no generated content appeared in this area). This could be because the parameter intervals are not wide enough to explore. It could also be because there is some kind of conflict between the metric features (which is the case here – we cannot have extremely dense levels which also have a lot of open space, so it makes sense that we can't maximise both metrics). Most importantly, however, it might mean that the generative algorithm itself needs revising, because its structure means that content of this type cannot be generated currently.

RERAs provide the user with information about the potential of the wider space their generator exists within, while ERAs confirm the current state of the generator. Using the two in tandem, the user can make adjustments to their generator and then verify the effect of the changes by performing regular ERAs and examining the changes made. This is an improvement on simply generating and examining single examples, because the increased density of data contained in an ERA or RERA allows the user to understand the shape of the generative space better. They can also hover over the histogram to view points of interest such as outliers, or more dense/sparse areas, to better inform their decision-making.

For example, Figure 3 shows a RERA with two different ERAs superimposed for the purposes of illustration. In this histogram, the y-axis shows *Wall Distribution* – a similar concept to Openness, which we described earlier and showed in Figure 2a – and the x-axis shows *Density*. The first ERA, circled at top of the histogram, shows that the generator was producing content with a high Wall Distribution score. The user has tried to reduce this, and the second ERA (circled in the lower part of the histogram) shows they have been successful in doing so. We can also infer other small changes from the shape of the second ERA – the spread of the generator's output has increased slightly, and the average density has been slightly reduced. The user may wish to make further changes if this is not exactly what they wanted.

IV. Automating Generator Optimisation

So far we have described the basic investigative and analytical techniques of Danesh. These provide data and feedback to the user, but are primarily user-led and only enable people to perform adjustments and exploration of the generator. The user is able to tag parameters to identify areas for investigation, adjust those parameters manually in the tool, and then run ERA and RERA analyses to explore the space. This process of iteratively changing parameter configurations and running

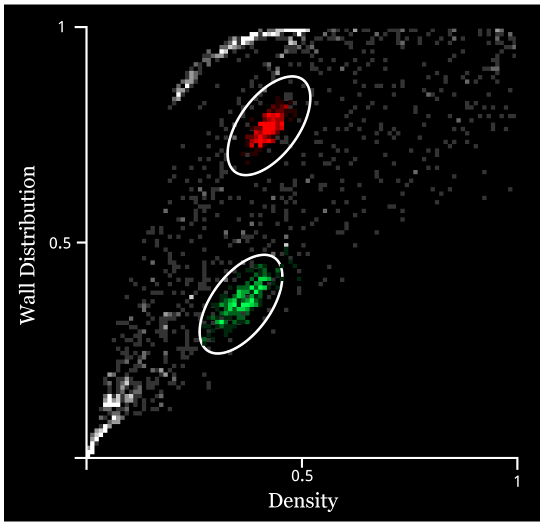


Fig. 3: An RERA with two ERAs superimposed (circled for B&W readers). The two ERAs show the state before and after adjusting the generator to reduce the y-axis metric score.

ERA analyses is time-consuming, however, and there may be hidden non-linear relationships between parameters or between parameters and metrics that make it hard to predict the impact of small adjustments on the output of the generator.

In this section, we describe features implemented in Danesh to automate parts of this process, and report on some analyses of the approaches we undertook. In the future we intend to provide automation support for other tasks the user may wish to undertake with the tool – we discuss these in Future Work.

A. Automatic Parameter Identification

Recall that in order to add a generator parameter to Danesh's interface the user must add C# Attribute tags to their code to label each parameter individually. Adding parameters indiscriminately is not a good idea, since it clutters up the user interface and makes automation tasks more difficult. It can also confuse novice users who might need help focusing on parameters that have meaningful impact on the generator. Thus, deciding which parameters to add and which to leave out is a difficult decision to make. There are also many reasons why a user might mistakenly decide not to add a parameter to Danesh – perhaps they forgot the parameter was there, underestimated how useful it might be, or simply don't want the burden of deciding which parameters to add. It would be helpful for the user if Danesh were able to intervene and automatically add useful generator parameters. To facilitate this, we have implemented a feature into Danesh which searches for fields in the generator code, tests their feasibility for influencing the generator, and then prompts the user to add them with upper and lower bounds and a name. The process by which a particular field is tested for feasibility is as follows.

First, before testing any fields, Danesh collects an average metric sample by generating several pieces of content (currently we use a sample size of 20) with the generator as-is and then computing the average metric score and standard deviation for each metric across all the generated content. We then use a metaprogramming technique called *reflection* to examine the generator's class and extract a list of fields declared in it. For each of these fields (we currently only use numerical values and booleans) we set the field's value to a series of test values, generate another sample of 20 pieces of content, compute the average metric score for this new sample and compare it to the baseline metric average. If any of the test values for the field produce an average metric score more than one standard deviation away from the original generator average, we consider the field to have had an effect on the generator and add it as a potential parameter.

The testing values we use for the field are pre-set values chosen via earlier experimentation to cover common parameter ranges. For boolean values, we simply test both values and see if there is a difference. For numerical values we test the following values: $\{-1, 0, 1, -100, 100, \texttt{MAX\_VALUE}, \texttt{MIN\_VALUE}\}$. If the numerical type is a floating-point value we also test $\{-0.5, 0.5\}$. We currently do not test string fields or other types, this is planned for future work.

Once this process is complete, Danesh presents the user with a list of potential parameters to add, along with three input fields to set a name for the field, a minimum value, and a maximum value (the same information given when a user manually tags a field using an attribute in their code). The user can then either confirm the addition of the parameter to Danesh, or delete the suggestion.

As an example, when running this procedure on our example cave generator it suggests both the width and height of the cave as parameters to the system, as well as a boolean value which sets whether the edge of the map is considered solid or empty (for the purposes of calculating tile births/deaths). The width and height parameters are useful for scaling the system, while the boolean field has a large impact on the expressiveness of the system - setting the edges of the map to be treated as empty changes the generated caves greatly, making them much sparser and fragmented.

B. Parameter Configuration Search

One of the tasks Danesh was designed to facilitate is the act of changing a generator's parameters to achieve a different kind of output (in terms of the type of content being produced, how variable the content is, how likely outliers are to occur, what properties hold of the content, etc.). As we mentioned in the introduction, a small survey we conducted of 53 game developers indicated that 85% of them typically achieve this by adjusting parameters and then viewing individual output examples to assess whether the change was good or bad. We have already described in the previous section analytical tools in Danesh such as metric functions and expressive range analysis to help the user achieve some of these outcomes without the need for laborious output examination.

Danesh can also automate this process of parameter search entirely, allowing the user to simply specify a target outcome in terms of desired metric scores and then let Danesh search the parameter space to find a parameterisation of the generator whose average metric score is as close to the user target as possible. To set up an automatic parameter search, the user must provide some information: first, they select which parameters they wish Danesh to search over. Fewer parameters results in a more efficient search, but more parameters covers a wider space of possibilities, so the user must decide how broad they want the search to be. Then, the user selects which metrics they wish to target for the search, and what values they wish the average generator output to be.

We implemented three algorithms for searching for parameterisations, in order to test different approaches: random search (as a baseline), hill climbing, and evolutionary search. There are conflicting goals here when considering the best algorithm for parameter search. A high-quality solution is desirable, since we want to get a result as close to the user's specified goals as possible. However, the time taken is also an important consideration – Danesh is designed to be an interactive application, so long periods of time searching for a solution is not desirable. In this section we describe the algorithms and some experiments performed to assess the relative performance of each one.

All of the algorithms use the same definition of fitness to assess the quality of a particular parameter configuration, which we define as follows. Given a set of n metric functions $f_1 \dots f_n \in F$, a target value t_i for each metric function $f_i \in F$, and a set of p generated content samples $c_1 \dots c_p \in C$, we define the **fitness** of a parameter configuration as follows:

$$m_i = \frac{\sum\limits_{c_j \in C} f_i(c_j)}{|C|} \qquad \delta_i = |m_i - t_i|$$

Where m_i denotes the mean value for metric function f_i on the set of generated content C, and δ_i denotes the difference between the user's target value t_i for the metric and the observed mean value m_i. The score Φ is then expressed as an average of these differences:

$$\Phi = \frac{\sum\limits_{i=1}^{n} \delta_i}{|F|}$$

Randomised search randomly generates parameter values for the selected parameters, using the minimum and maximum values set by the user as limits. It then evaluates them by sampling from the generator and recording the average metric value for each metric. The random search terminates after a set number of iterations, at which point the best parameter set seen is returned. It can also terminate after a set amount of time has elapsed. The number of samples per iteration and the number of iterations are parameters to the system – in the experiments here we test 15 samples over 100 iterations.

Hill climbing randomly generates an initial set of parameter values, and then iteratively changes one of the parameters by a small fixed interval, ensuring that each time it picks the interval change that results in the biggest increase in fitness. If it cannot increase its metric score any further, it has reached a local maxima. It records this maxima if it is higher than any it has found so far, and then randomly restarts. The hill climber terminates after a set number of iterations, or after a set amount of time has elapsed. As with randomised search, the number of samples per test and the number of iterations can be set as parameters to the system. In the experiments outlined here we use 15 samples per test, over 100 iterations.

Evolutionary search generates a population of random sets of parameter values and then performs an evolutionary search, crossing over parameter sets using one-point crossover on the parameter array and mutating a parameter value with a 5% probability. It terminates after a fixed number of generations have been completed, but can also terminate at the end of a generation cycle after a time limit expires. We use a population of size 15 evolved for 15 generations for these experiments.

1) Experimentation: We set up three auto-tuning scenarios of varying difficulty to test the algorithms, based on the example cave generator:

- **P1:** 2 target parameters, 2 target metrics.
- **P2:** 2 target parameters, 3 target metrics.
- **P3:** 4 target parameters, 4 target metrics.

Because responsiveness is important to the design of the Danesh tool, in our first experiment we were interested in the performance of each technique in a time-limited scenario. We ran each algorithm on each problem five times, and on each run we recorded the best fitness available at 2.5 seconds, 5 seconds, 10 seconds and 20 seconds. These were considered hard limits on time, so if the algorithm was computing an iteration at a time limit, we recorded the best fitness reported so far. Figure 4 details the results for each algorithm and problem case combination.

There are a few points worth making about the data available. First is that, in general, there is not a huge disparity between the three algorithms at the 20s mark. However, we can see that in many of the cases the hill climbing and evolutionary search algorithms perform badly at shorter timing marks – on P2 and P3, the evolutionary algorithm fails to complete processing a generation before the 2.5s mark. As these two algorithms progress, they improve in larger amounts. We believe that the primary reason for this disparity is the number of samplings and evaluations required to iterate the algorithm. Random search repeatedly chooses and evaluates new parameter configurations - meaning it can try several configurations across a wide search range in a few seconds. The evolutionary approach evaluates fifteen different configurations in each generation, however, and the number of evaluations the hill climber makes scales with the number of parameters (since it checks incremental changes to each parameter individually). This means that initial progress is slow, but rapid improvement is made once the algorithms progress.

To highlight this, we ran a second experiment which focused on minimum fitness. Instead of sampling fitness at preset timing cutoffs, instead we tested how long each algorithm took to reach a fitness of 0.9 and 0.95. This is to simulate Danesh finding a 'close enough' result, from which a user

	2.5s	5s	10s	20s
RS	0.892	0.909	0.959	0.964
HC	0.784	0.959	0.959	0.963
EVO	0.932	0.952	0.966	0.967

(a) Timing results for P1.

	2.5s	5s	10s	20s
RS	0.868	0.879	0.892	0.909
HC	0.738	0.806	0.892	0.911
EVO	-	0.887	0.897	0.917

(b) Timing results for P2.

	2.5s	5s	10s	20s
RS	0.889	0.906	0.916	0.942
HC	0.851	0.914	0.932	0.948
EVO	-	0.877	0.925	0.942

(c) Timing results for P3.

Fig. 4: Results showing best fitnesses recorded at fixed time intervals on three problem scenarios. RS: Random Search, HC: Hill Climb, EVO: Evolutionary Search. All results to 3 significant figures, an average of five samples.

	> 0.9	> 0.95
RS	2.61	4.29
HC	2.71	2.21
EVO	3.35	5.65

(a) Fitness-limited timing results for P1.

	> 0.9	> 0.95
RS	11.3	-
HC	11	-
EVO	11.6	-

(b) Fitness-limited timing results for P2.

	> 0.9	> 0.95
RS	2.06	30
HC	5.61	17.3
EVO	10.4	17.9

(c) Fitness-limited timing results for P3.

Fig. 5: Results showing time taken to reach a fitness of 0.9 and 0.95. RS: Random Search, HC: Hill Climb, EVO: Evolutionary Search. All results to 3 significant figures, an average of five samples.

could conceivably perfect or tweak the details to fine-tune the result. If the algorithm did not reach the fitness limit in 60 seconds, we recorded a failure. The results are shown in table 5 for each algorithm and problem case combination, as before.

First, note that the 0.95 target was not reached on P2 for any of the algorithms - this is because the targets set could not be reached closely enough in this problem scenario (in that it was slightly outside of the generator's expressive range). P2 is the hardest problem of the set because although it has fewer metric targets than P3, it also has fewer parameters it can change to reach those targets. The most important result here is that while the results are relatively close again, both hill climbing and evolutionary search far outperform the random search on the 0.95 fitness target for the hardest problem, P3. This shows that although random search can rapidly explore the state space to find relatively good results, the more intelligent techniques work better when prioritising high fitness.

We are still developing and refining techniques for automatic parameter configuration, but we believe that a hybrid approach may yield good results, where the first 5-10s is spent using random search, and then the best result is used to seed a hill climber. It is likely we will offer different techniques to the user depending on what tradeoff of speed versus quality they wish to have with the result.

V. Discussion

A. Current Limitations Of The System

We are currently working on various additions and improvements to the Danesh tool. Boolean and string fields are currently unsupported, and other conveniences such as numeric intervals or arrays are not supported in a convenient way (it is possible to use them in Danesh but workarounds are required). As we release Danesh to more developers and complete the user studies we are currently conducting, basic features like these will be implemented to round out Danesh's feature set.

Danesh's generality comes at the expense of placing a burden of implementation on the user. Currently, the user must write a visualisation method for their generator which makes it harder for novice users (although we provide a suite of utility functions to help with this, and text rendering requires almost no visualisation code). In the future, we hope to work on some automatic visualisation methods or a stock set of visualisers for common kinds of content (such as arrays representing tile-based levels, for example).

The other main implementation bottleneck for users is selecting and writing good metric functions. We plan to extend the automation of Danesh to cover this task, and provide other ways for users to express metrics, such as allowing the system to machine learn models of metrics. This will be conducted through an interface that allows the user to label positive and negative content examples and slowly define a metric function interactively. Good metrics are crucial to all of Danesh's more complex features, so this is an important area of future work.

Another potential limitation of the tool is that it is implemented as a plugin to Unity and written for C# and Javascript generators as opposed to being a general platform-agnostic tool. We do not see this as a limitation specifically, since we are primarily concerned with connecting with developer communities, and Unity is one of the most widely-used development tools today. Integrating with Unity and its asset store will help us contact developer communities directly and hopefully have a larger impact. However, we hope that the open-source nature of the tool will allow Danesh's techniques to be reimplemented into other languages, platforms and engines, should this be a barrier for other users.

B. Towards Domain-Agnostic PCG

The development of Danesh comes partly in response to a feeling that most work in procedural content generation, both within and outwith academia, is highly fragmented. Generative systems in games are typically highly bespoke, and as such it is harder to make theoretical connections between them. This is possibly one reason why Super Mario has continued to be such a common domain for procedural generation research – it is one area where there is a lot of overlapping existing work that provides baselines, inspiration and complementary sources of code. Of course, there are many other reasons to work in this domain as well – it is well-defined, the game is popular and well-known among potential survey participants, it is a well-understood design space. Yet the density of work provides additional appeal – comparisons, competitions, engines, tools.

While the fragmentation of generative software and research is a cause of its vibrance and diversity, it also hampers the formation of strong theoretical work that is independent of a particular domain, genre or game. While we do not see Danesh as a panacea for this, we hope that more projects like it that aim to be less domain-specific will help shift the targets of procedural generation research and encourage more domain-agnostic theoretical work on generative software. Open-sourcing Danesh hopefully enables interested researchers to branch off or extend Danesh with their own work, contribute features to the tool, and as we expand the library of example generators, will allow them to perform experiments across a wide range of generator and content types. This could contribute to more abstract theories of content generation for larger classes of games or design scenarios.

VI. Conclusions

In this paper we introduced Danesh, a tool for exploring, explaining and experimenting with procedural content generators. We described the basic functionality of the tool, and how it affords a richer way of visualising and interacting with generative spaces. We then discussed how Danesh can automate some aspects of its own process to greatly simplify the act of iteratively refining a generator's output. We evaluated several techniques for automating parameter configuration search, and then discussed the current limitations for the system and the potential future for domain-agnostic PCG tools.

We believe there exists a skills gap in game development concerning procedural generation, and that this gap is not being bridged by traditional tools. Writing procedural generators is already a difficult task, but understanding them well enough to tweak and adjust them to a designer's liking requires a lot of knowledge that is difficult to obtain. Other comparably complex (arguably even more complex) tasks such as writing graphics shaders have been made considerably easier thanks to intuitive and useful tools. We hope the same can be done for procedural generation, and that Danesh contributes towards this goal in some small way.

Procedural generation is often seen as a simple case of 'more unpredictable stuff', content that can be thrown into a game for endless replay value without much thought. But generative techniques are increasingly a key tool in achieving certain design goals, expressing artistic ideas, and developing new genres of game. In order to promote this growth and diversity, we need to support developers, students, dabblers and novices of all kinds, to ensure this technology is as flexible and accessible as possible.

Acknowledgments

The authors wish to thank Gillian Smith for insight into her expressive range analysis work. We also thank our reviewers who provided a lot of useful feedback for the project.

References

[1] D. Yu, "Spelunky," http://www.spelunkyworld.com, 2009.

[2] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 9 2011.

[3] K. Compton and M. Mateas, "Casual creators," in *Proceedings of the Sixth International Conference on Computational Creativity (ICCC 2015)*, Brigham Young University. Park City, Utah: Brigham Young University, June - July 2015, p. 228–235.

[4] S. Murray, "No Man's Sky," http://www.no-mans-sky.com, 2016.

[5] M. Treanor, A. Zook, M. P. Eladhari, J. Togelius, G. Smith, M. Cook, T. Thompson, B. Magerko, J. Levine, and A. Smith, "Ai-based game design patterns," in *Proceedings of the 10 International Conference on Foundations of Digital Games, FDG*, 2015.

[6] J. Togelius, N. Shaker, and M. J. Nelson, "Introduction," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2015.

[7] I. Parberry, *Introduction to Game Physics with Box2D*. CRC Press, 2013.

[8] W. T. Reeves, "Particle systems&mdash;a technique for modeling a class of fuzzy objects," *ACM Trans. Graph.*, vol. 2, no. 2, pp. 91–108, Apr. 1983. [Online]. Available: http://doi.acm.org/10.1145/357318.357320

[9] Unity Technologies, "Tree editor documentation," http://-docs.unity3d.com/Manual/class-Tree.html, 2016.

[10] S. Lague, "Procedural cave generation tutorial," https://u-nity3d.com/learn/tutorials/projects/procedural-cave-generation-tutorial, 2015.

[11] Captain Kraft, "Create a procedurally generated dungeon cave system," http://tinyurl.com/pcgtut, 2013.

[12] M. Ansari, *Game Development Tools*. Crc Press, 2011.

[13] M. Cook, "Make something that makes something: A report on the first procedural generation jam," in *Proceedings of the Sixth International Conference on Computational Creativity*, 2015.

[14] 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.

[15] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach." in *Proceedings of the Artificial Intelligence in Interactive Digital Entertainment Conference*, 2013.

[16] A. Liapis, G. N. Yannakakis, and J. Togelius, "Designer modeling for sentient sketchbook," in *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2014.

[17] M. Kerssemakers, J. Tuxen, J. Togelius, and G. N. Yannakakis, "A procedural procedural level generator generator," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 2012, pp. 335–341.

[18] R. Khaled, M. J. Nelson, and P. Barr, "Design metaphors for procedural content generation in games," in *Proceedings of the 2013 ACM SIGCHI Conference on Human Factors in Computing Systems*, 2013, pp. 1509–1518.

[19] M. Cook, "Generate random cave levels using cellular automata," http://tinyurl.com/pcgtut2, 2013.

[20] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 10.