Danesh: Helping Bridge The Gap Between Procedural Generators And Their Output

Michael Cook[1], Jeremy Gow[2], and Simon Colton[1,2]
[1]Metamakers Institute, Games Academy, Falmouth University
[2]Computational Creativity Group , Goldsmiths, University of London

ABSTRACT

Procedural content generation is more popular than ever among game developers, but understanding, adjusting and perfecting a procedural generator is difficult for newcomers and experts alike. In this paper we present *Danesh*, an intelligent tool we are building to help developers of all skill levels explore, improve and understand procedural generators. We discuss the structure of the tool, report on the techniques used, and lay out the future of the project.

Keywords

procedural generation, assistive design, computational creativity

INTRODUCTION

Procedural content generation (PCG) is an important part of modern game development technology (Mossmouth Games, 2008), useful both as a tool for solving problems and a paintbrush for expressing ideas, and often employed as both at once (Hello Games, 2016). The ability to generate game content automatically opens up the potential for new kinds of game to be made possible, as well as easing the development of traditional games by allowing abundance, serendipity and surprise to be added to a game in very simple ways. We are only just beginning to discover the potential of generative methods as a design tool.

Many other kinds of game development technique could be placed in this category, such as in-game physics. Physics engines were powerful tools for adding value to a game, making the world seem more real and chaotic. They eventually gave way to new mechanical ideas and ultimately entire games based solely on the notion of physics interactions, with Angry Birds being a particularly pure example of this. But there are important differences between technology like physics engines and procedural content generation, including that modern game development tools tend not to include any assistance for PCG built-in.

A tool such as Unity includes off-the-shelf support for pathfinding, basic AI routines, built-in customisable physics engines, a complete particle system, complex collisions in 3D or 2D space, and more – all of these are tools that speed up game development, and most importantly they provide access to these concepts for users who may not be familiar with the ideas behind them, or unable to write the code themselves. Most common PCG techniques do not appear in tools like Unity as default services, and tutorials tend to focus on single processes or highly specific kinds of content rather than generation as a broader concept, and so people approaching game development from any angle other than a programming background may find it harder to get started.

Additionally, implementation is only half the story. Building a procedural generation system only gets a designer so far – often there is a better technique to be used, a better parameterisation, a better way of achieving the intended results. Understanding the relationship between the generator they have written and the output it produces is extremely difficult. Many tutorials for procedural content generation focus on how to achieve a result, a complete algorithm that produces output. But the idea of refining this system and being able to adjust and tweak it to taste is often absent – (CaptainKraft, 2013) ends by stating 'the final step is down to you: you must iterate over what you learned to create more procedurally generated content for endless replayability'.

In this paper we present *Danesh*, a tool for exploring, understanding and improving procedural generators. Danesh is currently capable of visualising the expressive space of a generator, helping a user understand the effect of changing parameters, automatically adjusting parameterisations of a generator, and providing detailed analysis of content generated. Most importantly, Danesh can be integrated with new procedural generators using a simple programming interface, allowing it to automatically detect new generators and immediately start working with them.

In the remainder of the paper, we describe the tool in detail, discussing both the implementation of the tool itself and the functionality it provides. In *Related Work* we introduce some background in procedural generation analysis that informed the work in Danesh, particularly with respect to expressive range analysis. In *Danesh* we describe the basic functions of the tool and how it is used within Unity. We also describe the techniques used and go into depth about the tool's auto-tuning function. In *Future Work And Open Problems* we describe the plans for the tool's ongoing development, and argue how these are also important objectives for procedural content generation research as a whole.

RELATED WORK

In this section we provide an overview of some work relating to procedural content generation either in the context of it being studied as a tool for designers, or methods for evaluating generators in certain ways. For a full overview of AI research in procedural generation, we direct the reader to (Togelius et al, 2011).

In (Khaled et al, 2013) 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 for 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 (Khaled et al, 2013) 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.

In (Craveirinha et al, 2013) the authors propose a framework for assisting designers in experience-driven procedural content generation, and report on a preliminary experiment to optimise the Infinite Super Mario level generator by attaching a genetic algorithm to several parameters of the game. The generator is then told to optimise for a particular player outcome such as the average number of times a player dies. The paper mostly proposes a framework for future development, since their tool is hardwired to one particular generator in one particular game, and dependent on having parameters provided that the designer knows will have an impact on achieving their stated design goals. Our work with Danesh stands apart from this partly by prioritising generality and interactivity, but also because it requires less design knowledge up-front, and can be used as an exploratory tool as well as a targeted problem-solver.

The Sentient Sketchbook, described in (Liapis et al, 2014), is a similar design assistant that uses procedural generation. Instead of optimising a generator, the Sentient Sketchbook is a combined generator and co-creator, taking on responsibility for the design of maps and levels and able to engage a designer in a kind of dialogue. The Sentient Sketchbook's emphasis on visualisation inspired the development of Danesh's interface, although the tools are quite different – the Sketchbook is focused primarily on map generation and is built into its own generator, whereas Danesh is intended to be a general-purpose tool and can be combined with user-written generators or evaluatory metrics (although this has not been fully implemented at the time of writing, the architecture is present within the tool).

Expressive Range

One of the important functions in Danesh currently is the ability to analyse the expressive range of a generator, and then use that information to perform further analyses and adjustments to the generator. The notion of analysing a procedural generator's expressive range is derived from (Smith and Whitehead, 2010), which describes a tool, *Launchpad*, which generates levels for a platforming game. The authors outline their analytical approach to evaluating expressive range as the following process:

- **Determine appropriate metrics** – a set of metrics for evaluating different qualities of one piece of generated content, in Smith and Whitehead's case a platformer level. They write: 'These metrics should be based on global properties of the levels, and ideally should be emergent from the point of view of the generator'. We believe that by this the authors mean that these metrics should return meaningful values for any valid level (rather than only analysing levels with a particular piece of content in, for instance) and that the metric value should be dependent on the output of the generator, and not be something that could be calculated based simply on an inspection of the generator's code.
- **Generate content** – a large number of pieces of content are made by the generator and collated together. The metrics are applied to each piece of content to score it.
- **Visualise generative space** – in this step, the metric scores for the generated content are visualised in some way. Smith and Whitehead propose 'one effective way to view

this range is by creating a number of 2D histograms, where the axes are defined in terms of the metric scores'.

- **Analyze impact of parameters** – by analysing the histograms, one can now identify anomalies in the generative space according to the metrics defined in the first step.

One of the motivations behind this first version of Danesh was to take this process and make it extensible and easily accessible to less experienced users. However, we will also show how Danesh extends the use of expressive range analysis so that it becomes a jumping-off point for further analysis work and improvement of the generator.

DANESH

In this section we will describe the operation of Danesh, its various functions and how they are implemented. We conclude with a brief explanation of how new generators are integrated into the tool, to show how we are aiming to make the process as simple as possible. Danesh is written in C# as a plugin to the Unity game development environment. Although the techniques we describe in this paper could easily be reimplemented in other libraries and platforms, we selected Unity because of its popularity and its ease of extension. Danesh is open source and can be downloaded via GitHub[1]. Tutorials for its usage are also available online[2].

Before Startup

Before we describe Danesh's appearance and functionality, we wish to give the reader some background on how the tool loads information and presents it to the user. One of the important features of Danesh is its generality and its ability to load in unseen procedural generators and still be able to provide analysis and assistance in exploring the generator. It achieves this through the use of reflection, a metaprogramming technique that allows software to examine code at runtime and extract information from it. This means that there is a small amount of setup required to make a procedural generator compatible with Danesh, but we are working to make this as minimal as possible.

Danesh needs some initial setup before it can load a procedural generator – it needs access to the generator itself, meaning a segment of code it can execute that results in a piece of generated content; it needs a visualiser that can turn generated content into something displayable; and it needs a list of parameters that affect how the generator runs. It discovers these things using C# *attributes*, a way of labelling code that can be discovered at runtime. The following code shows a method that has been tagged with an attribute called `[Generator]`:

```
[Generator]
public Tile[,] GenerateLevel(){
        //Generator code...
        return tiles;
}
```

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

This attribute should be placed on the method that generates the content to be analysed. It needs to have a very specific method signature: it should take zero arguments and return an object of some kind. Danesh is agnostic to what the object is or what it contains - all methods that interact with the object, such as the visualiser, have their domain-specific code provided by the user currently.

The visualiser is labelled using a `Visualiser` attribute. The user can either provide code which renders something onto a texture –useful for displaying visual content such as maps –or a much simpler text renderer. Even if the content is not explicitly textual, it may make more sense to display it this way. For example, an item generator for an RPG might render its output as a list of details about the item.

Parameters are tagged to indicate that the user wishes to interact and adjust them using Danesh. Unlike the `[Generator]` attribute, this attribute has additional metadata provided. The following code shows a parameter tagged with the `[Tunable]` attribute.

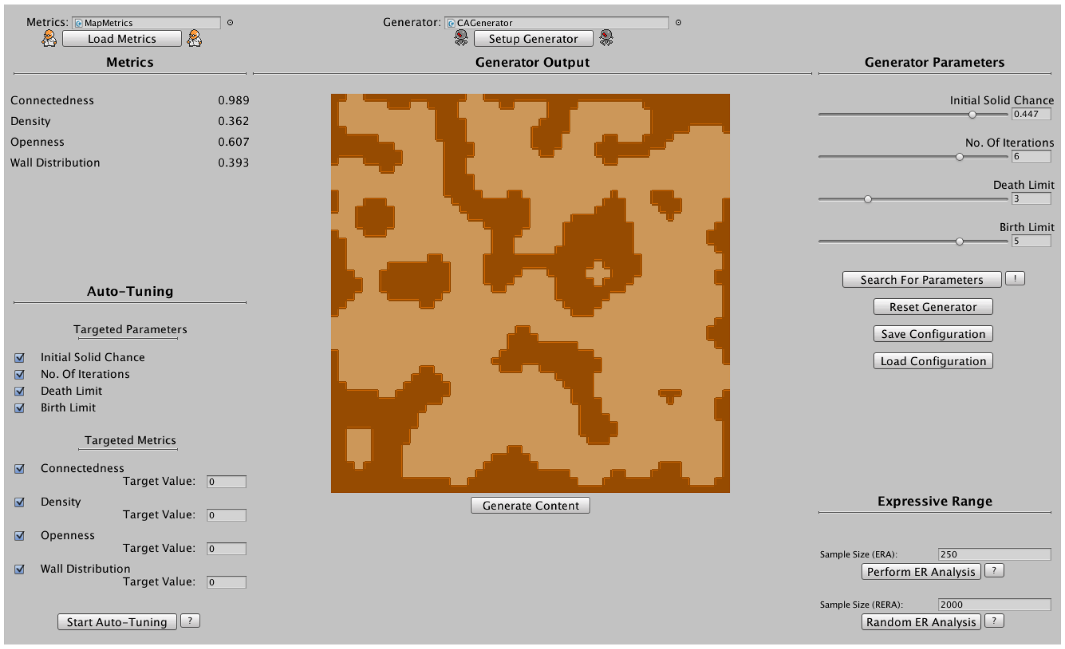
```
[Tunable(MinValue: 0.1f, MaxValue: 0.7f, Name:"Initial Spawn Chance")]
public float ChanceTileWillSpawnAlive = 0.45f;

[Tunable(MinValue: 1, MaxValue: 6, Name: "No. Of Iterations")]
public int NumberOfIterations = 5;
```

`MinValue` and `MaxValue` are used by Danesh to understand the limits of the parameters. These can be set by the user to limit Danesh's exploration of the parameter space to keep it within limits that are likely to be interesting, or to avoid values that would cause extremely long execution times or crashes. Of course, if the user is unsure about what interval would be useful to examine, they can set very large values and let Danesh work harder. These parameters are largely for fine-tuning or extra customisation. `Name` is used in Danesh's UI to provide an easier way of seeing what a parameter is for, rather than using its variable name in code.

Danesh is implemented as a plugin to Unity, meaning it runs as a separate editor tab in the main workflow of the tool. When it runs, the user specifies a generator object in the open scene, and Danesh uses C#'s reflection library to inspect the code. One of the features of reflection is to search for custom attributes such as our `Generator` and `Tunable` attributes. Once it finds these, it launches with the tagged generator method and parameters loaded in.

Figure 1 shows a screenshot of Danesh just after starting the tool up. Danesh currently provides several default procedural generators already configured for use with the tool, meaning they can be run without writing any code in order to experiment with the tool. The generators include basic versions of the popular Drunkard Walk algorithm (Roguebasin, 2014) and a Cellular Automata Generator (Cook, 2013), with both generators producing two-dimensional arrays representing game levels. We discuss the expansion of the tool beyond two-dimensional level generators later in the paper. The following sections describe Danesh's functionality after loading.



Metrics: MapMetrics

Generator: CAGenerator

Metrics

Connectedness	0.989
Density	0.362
Openness	0.607
Wall Distribution	0.393

Generator Parameters

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

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

Sample Size (ERA): 250

Sample Size (RERA): 2000

Figure 1: The main screen of Danesh at startup.

Generating Content

The 'Generate Content' button in the bottom-centre of Figure 1 generates a single piece of content from the currently selected generator, and displays it in the centre of the screen. It does this by calling the method tagged with the `Generator` attribute, and then passing this object to the `Visualiser` method. This can be repeated many times to see examples of content in rapid succession. Each time a piece of content is generated, the currently active metrics (which we will cover later) are calculated for the new content and displayed in the top-left corner. This provides an immediate connection between generated content and metric for the user.

Parameter Control

In the top-right corner is a list of parameters that control the behaviour of the generator. This is a list of all parameters tagged with the `Tunable` attribute, displaying whatever name was given in the attribute's metadata. Changing the values of the parameter immediately feeds back to the underlying generator, which means that pressing the Generate Content button will take these changes into account, and generate a different kind of content to before. Danesh currently supports numerical parameters such as `float` and `int`, as well as `boolean` values which have their own interface for changing their value (colour-coded to show True and False settings). Other types, including lists and more complex data types, will be added as the tool develops.

Metric Measures

In the top-left corner are a set of metric scores, based on the same concept described by (Smith and Whitehead, 2010) earlier. A metric captures some quality about a piece of generated content, and scores it in the numerical interval $[0, 1]$. Because Danesh does not know in advance what kind of content is being generated, the user currently is responsible for writing their own metrics and tagging them with `Metric` attributes so that Danesh can discover them. Deciding what a useful metric would be and writing it is the most complex part of using Danesh currently, and we will expore ways to make it easier to do in future.

In this paper, examples will refer to the Openness and Density metrics for maps. Openness measures how empty a level is – it calculates the number of tiles that have no solid neighbours, and returns that number as a percentage of the total map size. Density measures the total number of solid tiles, and returns that as a percentage of the total map size. Note that these two metrics are related, but not linked – a map can be low density but not be very open, if the open space is broken up with lots of solid walls, for example.

Expressive Range Analysis

These metrics are also used in the expressive range analysis (ERA) feature, which is situated in the bottom-right of the screen. As described in the section *Related Work*, Danesh creates a two-dimensional histogram for visualising the results of an ERA pass, with the two metric values providing each data point. When the user calculates an ERA, Danesh samples the generator many times and records metric scores for each piece of content. When complete, the user can then select two metrics from drop-down boxes and see a histogram generated for them.

There are two buttons in the ERA section, labelled 'ERA' and 'RERA'. The ERA button performs an expressive range analysis using the *current* parameterisation of the generator. This is closest to what is described in (Smith and Whitehead, 2010). It runs the generator a large number of times to accumulate many metric scores, and then displays the results in a histogram with each axis referring to one of the two selected metrics. Currently it generates 2,000 pieces of content for the histogram. (Smith and Whitehead, 2010) use 10,000 for their expressive range analysis, but we found 2,000 to be an effective tradeoff given the interactive nature of the tool. Performance and speed is something we discuss later in the paper, and we intend to make this value customisable in future versions of Danesh. Once the ERA is complete, the histogram displays in the center of the screen. A darker spot on the histogram indicates that fewer pieces of content had those two metric values, while a lighter spot indicates that more content fell into this area. Figure 2 shows a sample ERA of the default Cellular Automata Generator.

RERA stands for Random Expressive Range Analysis. This runs the generator many more times than a standard ERA, but each time it *randomises* the values of the parameters of the generator using the minimum and maximum values provided in the `Tunable` attribute metadata. The resulting ERA histogram may look very different to the standard ERA, as it shows the spread of metric values across a large number of possible parameter settings. In doing so it reveals how different parameterisations of the generator result in different expressive spaces. The RERA is a useful tool for showing the potential unexplored space inside a procedural generator, since it reveals to the user whether changing parameters can move the generator to a new subspace of the multi-dimensional metric space. This might spur the user on to adjust parameters and tweak the expressivity of the generator. Figure 3 shows a RERA of the same generator as Figure 2, with the original ERA painted in red (circled for those reading in black and white).

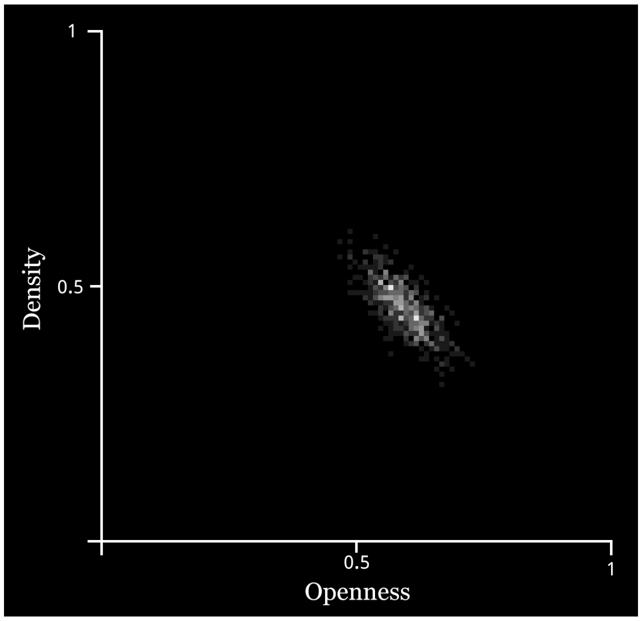


Figure 2: An expressive range analysis of a cellular automata level generator.

Auto-Tuning Parameters

The final feature of Danesh brings together all of the other aspects of the tool into one function called *auto-tuning*. The expressive range analysis histograms reveal useful information about a procedural generator, and by using a RERA the user can discover that there are new expressive spaces that their generator could occupy but that they have not been able to parameterise it for yet. While the user could change parameters manually and try to run ERAs to discover if they are successful, this is a slow process and may not always work (since a linear change in a parameter may not result in a monotonically linear change in metric values, and parameters may not be linearly linked).

Ideally, we would like the user to be able to indicate a region of the expressive range space they are interested in and have Danesh automatically seek out a parameterisation that results in a similar expressive range histogram. Auto-tuning attempts to provide this, by allowing Danesh to automatically vary parameter values to search for settings that result in particular metric values or regions of the expressive space.

In the bottom-left corner of the main screen there is a list of metrics, derived from the same attributes that generated the metric reports and the ERA metric selections. Each metric has an input field next to it and a checkbox. The user can write in the values they'd like to target for each metric, and check the box to indicate that that metric should be included in the computational evolution. This allows the user to target only a single metric, or to target multiple simultaneously.

Before the process begins, the user can also tell Danesh which parameters it is allowed to change in order to achieve the metric output. Even if the user wants help from Danesh in tuning its generator, they may already have some insight into which parameters are likely

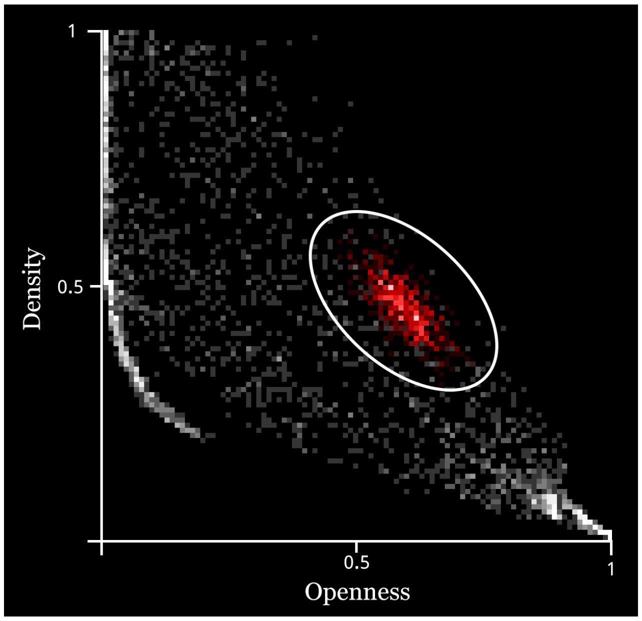


Figure 3: A random expressive range analysis (RERA) of a cellular automata level generator. The ERA from Figure 2 is overlaid in red and circled.

to be changed, or they may simply not want Danesh to change certain parameters because they have already been tuned in a way the user is happy with. Unchecking boxes next to parameters in the top-right corner will stop Danesh from changing this parameters during the auto-tuning process.

Searching For Parameter Values

When the user clicks 'Auto-Tune', Danesh begins a search for parameter values. We tested four techniques for performing this search: an evolutionary system, a grid search, a hill climber, and a random search. Before we discuss the use of these techniques, we briefly describe how each one works in turn. In all cases, a set of parameters is evaluated to assess how good it is at achieving the user's intentions. The fitness measurement for a set of parameters is based on an average of the distance between the target metric values set by the user and the metric values measured on the generated levels. Formally:

Given a set of n metric functions $f_1 \ldots f_n \in F$, a target value t_i for each metric function $f_i \in F$, and a set of generated content $c_1 \ldots c_p \in C$, we define the fitness of a parameterisation of a generator 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 for the metric and the observed mean value. The fitness Φ is then expressed as an average of these differences:

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

Evolutionary Search Each member of the population is a list of parameter values, with one for each selected parameter in the UI. A member is initialised by randomly setting the value between the minimum and maximum values set by the attribute metadata. Danesh then evaluates the population by taking a set of parameters, applying them to the generator, executing the generator a number of times and recording an average score for each of the target metrics, as described earlier.

After the population has been sorted according to fitness, the highest-fitness parameter lists are crossed over to create a new population. The top half of the population are used to generate the next population. Crossover is performed by one-point crossover on the list of parameters, with a 5% mutation rate (derived after some experimentation) which inserts a new random value for a parameter instead of inheriting from its parents.

Danesh's default settings run a population of 20 parameters lists, run for 20 generations, with 70 examples generated for each evaluation of a parameter list. Like the expressive range analyses, we found these values to be a useful tradeoff of evolutionary thoroughness and a reasonable execution time. Despite this, using evolution for auto-tuning is still a slow process and can be even slower if the generator takes a long time to generate a single piece of content. We will discuss this later in future work as an ongoing issue.

Grid Search Inspired by its use in parameter optimisation in machine learning (Pedregosa et al, 2011), we implemented a grid search which attempts to evenly sample the parameter space. Given a sampling rate s, we construct sets of parameter values such that every parameter is set to s different values in the value range (that is, each time we change the parameter, we change it by $1/s$ of the difference between its maximum and minimum value). We test every permutation of these parameters, such that for p parameters and a sampling rate s, we generate s^p sets of parameters.

Hill Climbing We also implemented a random hill climber which resets when it reaches a local maxima but remembers the maxima in the case that it does not find a better example. The hill climber terminates after a certain amount of time has passed. This technique is specifically designed to offer a tradeoff of performance against time, by being able to provide an upper bound on its running time.

Random Search As a comparator, we implemented a random search which performs as many evaluations as the Grid Search by repeatedly generating and testing random parameterisations.

Technique Evaluation Results

There are two important measures to consider in evaluating techniques for auto-tuning. The first is how well the auto-tuning performs; that is, the 'fitness' of the best parameter set found during auto-tuning (we use the term *fitness* across all three auto-tuning techniques despite it being a term associated with evolution). The user is looking for the parameter set that most closely fits their target metrics, so a higher score is desirable. We must also take into account a second measure for auto-tuning, however, which is the time taken to run. We initially implemented evolutionary search but despite its good performance we were concerned that it was taking too long to perform a search that might not require such complex techniques.

To provide insight into the effectiveness of the three techniques we have described we ran a short experiment on two different auto-tuning problems, one targeting a single metric and one targeting two metrics. For the evolutionary search we used a population of size 20, run for 20 generations with 70 examples generated per run. For the grid search we used 4 samples (for a total of 256 evaluations) and for the random search we used 256 evaluations to match the grid search. The results of these tests are presented in Figure 4.

	Fitness (1 Metric)	Time (1 Metric)	Fitness (2 Metric)	Time (2 Metric)
Evo.	0.914	229.767	0.938	408.195
Grid Sch.	0.909	94.44	0.948	96.652
Hill Climb	0.957	70	0.9402	70
Random	0.905	92.573	0.924	92.35

Figure 4: Results from preliminary auto-tune testing. Evolutionary, Hill Climb and Random data are averages of five runs (Grid Search always returns the same result). Time is measured in seconds.

We can see that fitness is high for all techniques (fitness is measured in the range [0,1] with 1 being the best possible parameterisation), but that the differences between more complex techniques and a hill climber or similar random search are not large - although random does slightly underperform in all cases. We believe there are good reasons for the other two techniques not performing as well, however. Grid search is a very broad-strokes approach to parameter optimisation and the best values may often lie between grid settings. Higher values for the sample rate would work better, but the technique is unable to filter out obviously useless areas of the search space. It's understandable that in this scenario, a random sampling of the space at the same rate might yield better results. This will greatly depend on the maximum and minimum settings for each parameter - if they are set too widely, then a grid search spends a lot of time searching useless areas of the parameter space.

We believe that the evolutionary approach is affected by the fact that the fitness evaluation is an *average* of the output of the generator, which only approximates the centre of the generator's space. This space can be large, however, and produce content with widely different metric scores within the same parameterisation. If the evaluation does not check a large enough sample of the generators output, it might estimate the average incorrectly and cause the evolutionary system to move away from fitter areas of the parameter space. This is largely dependent on the generator and metrics in question - some generators have large variance in the metric qualities of their output.

The hill climbing approach produces consistently high fitness, and has the additional benefit of being able to cleanly terminate (often early, since we can optionally terminate if a fitness is found above a certain threshold). We've implemented this as the main auto-tuning technique for now. After user studies and more generators are tested we may revisit our approach here.

Sample Parameterisations

Before we conclude this section, we provide some examples of different parameterisations of the Cellular Automata Generator after different auto-tuning runs. Note that two different parameterisations of the generator may result in identical or similar metric scores, so these are only representative of a single example of the generative space. Figures 4-7 show three samples each from four different auto-tuned generators, with examples of both single and double metric targeting. Figure 4 produces open arena-like levels with some corners and hiding spots. Figure 5 targets lower openness which results in more passages and winding corridors and less open space. Figure 6 and 7 show more extreme parameterisation, with the latter in particular being an interesting example of Danesh producing unusual outputs. Low openness and low density are somewhat opposed to one another, but the tradeoff Danesh discovered results in maps with small amounts of single tiles (resulting in low density), which breaks up open space (resulting in low openness).

FUTURE WORK AND OPEN PROBLEMS

With Danesh, we aim to build a platform where procedural generators of all kinds can be loaded, analysed, understood and improved in many different ways. There are three key goals we have in mind when developing the tool further: *accessibility*, ensuring Danesh has as low a barrier to entry as possible; *power*, expanding the capabilities of the system and enabling it to provide new functionality; and *generality*, to build an adaptive tool that can analyse generative systems regardless of their output types or target domains. Currently this leads us to a few specific points of future work that we outline in this section.

Efficiency Tradeoffs

Analysing a procedural generator requires sampling from it frequently, and some of the techniques we put forward here such as auto-tuning require sampling a generator thousands of times. Even thousands may not be sufficient to obtain good results. Typically a procedural generator can output content in a fairly short time, but even a generation time measured in hundredths of a second can result in many minutes of waiting to perform a simple analysis in Danesh. The user can already adjust the size of the analysis in order to change how thorough it is. Longer-term solutions, however, rely on the efficiency of the generator itself and that may not be possible to circumvent or improve. It's possible that building a cached repository of generated output for a parameterisation might save time. Keeping Danesh efficient and quick to use is an important open problem area.

Metric Catalogues & Automation

Users can already easily add their own metrics to Danesh, and we hope to expand a suite of provided metrics for common content types or analysis, to give support to users who may not know what kind of information will be useful to them. This in itself is a large future work problem for procedural generation research as a whole - what metrics are useful in analysing generators of certain types of content? How generally applicable are these metrics across many different algorithms or domains?

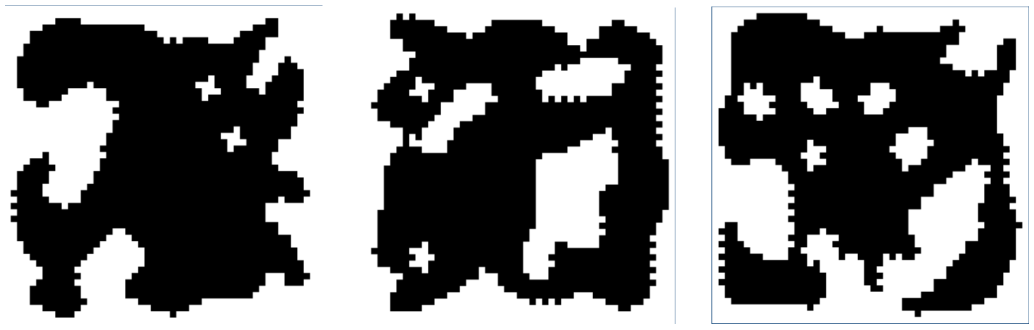


Figure 5: Samples from a parameterisation targeting openness of 0.8

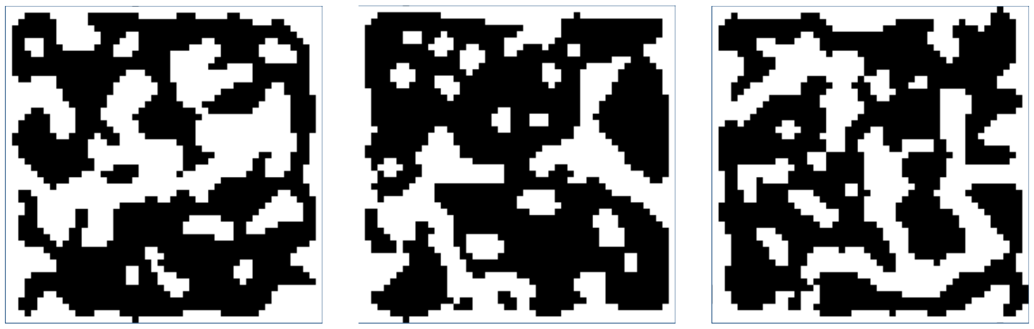


Figure 6: Samples from a parameterisation targeting openness of 0.5 and density of 0.5

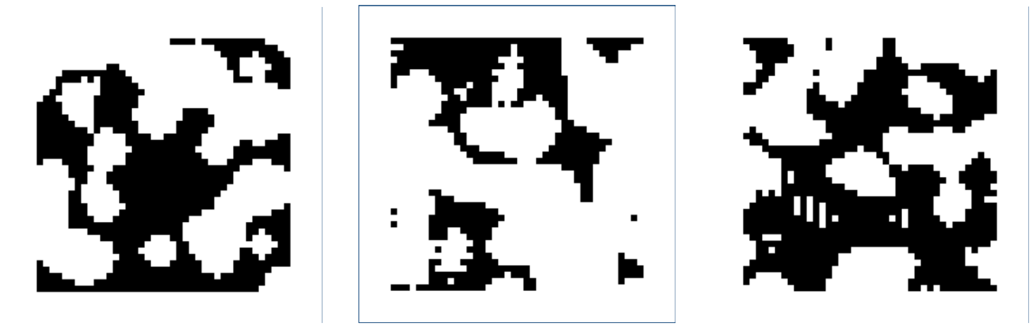


Figure 7: Samples from a parameterisation targeting density of 0.8

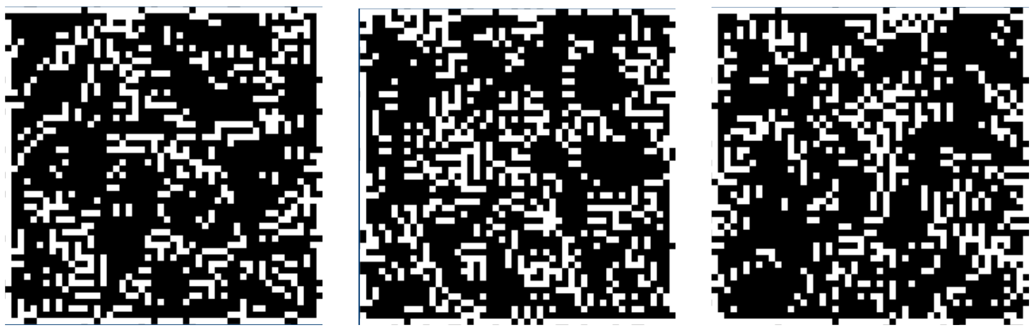


Figure 8: Samples from a parameterisation targeting openness of 0.2 and density of 0.2

Once a large quantity of default metrics are available, we hope to extend Danesh to be able to automatically apply metric evaluations to a generator and estimate which metrics offer the most insight or utility for analysing a particular generator, possibly employing ideas from information theory (Shannon, 1948) in order to assess metric worth. This would allow Danesh to propose metric selections for initial ERAs and perhaps automatically guide the user through the tweaking and improvement process, to identify new areas of the generative space and how best to move through them.

Generative Space Visualisation

The 'Generate Content' button currently is the user's only way to view the output of its generator under a particular set of parameters. Although metrics and ERA analyses can help the user understand what a generator is doing, it is also crucial to be able to see the results of the generator to provide context for these other indirect measurements of the generator's performance. Generating a single map at a time is useful as a guide, but we believe it is important to provide a richer, higher-level method of visualisation.

In related problems such as evolutionary art applications, the visualisation of generator output is achieved by showing a sample of the generator's output in a grid, allowing the user to absorb multiple examples of a generator simultaneously and be able to mentally note similarities and differences that indicate what space the generator is currently occupying. We intend to implement a similar function to Danesh, to complement the current ability of the user to hover over and preview examples from the ERA histogram. We believe that relating the metrics to example outputs more directly will help the user build a model for what each metric means in relation to the particular generator they are using.

Automating Parameter Extraction

As a more experimental point of future work, we wish to explore the idea that Danesh can help the user even with the configuration of Danesh itself. Using a generator with Danesh will eventually require the tagging of metrics, parameters, visualisation methods and the generate method itself. While some of these (such as metrics) can be provided by Danesh, other elements which are crucial to the analysis (such as the parameters) must be identified by the user before loading Danesh and tagged with the `Tunable` attribute.

We believe that it might be possible for Danesh to propose new parameters based on a metaprogramming analysis of the generator's code. On a simple level, Danesh can identify public fields within the code and test out each one as a possible parameter whose value can be varied to change the generator. On a more complex level, Danesh might be able to identify common use of numbers (sometimes referred to as the 'magic number' code smell (Martin, 2009), where numbers are inserted into code without variable names or any explanation) and then extract this value out of the code in multiple places, generalising it into a parameter. This is quite a tall order and may be infeasible for most generator cases, but it is certainly an interesting problem to try and solve, as it would allow Danesh to make suggestions to novice users and automatically identify useful ways it can adjust or change a generator. This remains a fairly blue-sky area of future work, however. Our initial experimentation with this suggests it is possible for some simple cases but a difficult problem in general (with unpredictable side effects for the user).

CONCLUSIONS

In this paper we presented *Danesh*, a Unity tool that can be used to inspect, adjust and analyse procedural generators. We discussed the motivation and plan for the tool, and described an early version of the software. We showed the current state of the tool's functionality, and how this will be expanded in the future, as well as other interesting problems we hope to examine along the way.

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

Thanks to Gillian Smith for many discussions of procedural generation and expressive range. Many thanks to Adam Summerville for supplying the LaTeX template for this paper.

BIBLIOGRAPHY

CaptainKraft. Create a Procedurally Generated Dungeon Cave System. Published by TutsPlus, available online at `http://tinyurl.com/pcgtut`

Cook, M. Generate Random Cave Levels Using Cellular Automata (2013) Published by TutsPlus, available online at `http://tinyurl.com/pcgtut2`

Craveirinha, R., Santos, L., Roque, L. An Author-Centric Approach to Procedural Content Generation. In Proceedings of the 10th International Conference in Advances in Computer Entertainment Technology, 2013.

Hello Games (2016) No Man's Sky [PC, PlayStation 4] Hello Games.

Khaled, R., Nelson, M., and Barr, P. Design Metaphors for Procedural Content Generation in Games. In the Proceedings of CHI '13, 2013.

Liapis, A., Yannakakis, G. N., Togelius, J. Designer Modeling for Sentient Sketchbook. In Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG), 2014.

Martin, Robert C, (2009). "Chapter 17: Smells and Heuristics - G25 Replace Magic Numbers with Named Constants". Clean Code - A handbook of agile software craftsmanship. Boston: Prentice Hall.

Mossmouth Games (2008) Spelunky [PC] Mossmouth Games.

Pedregosa et al. Scikit-learn: Machine Learning in Python. JMLR 12, pp. 2825-2830, 2011.

Roguebasin. Random Walk Cave Generation. (2014) Available online at `http://tinyurl.com/pcgtut3`

Shannon, C. E. A Mathematical Theory of Communication. In The Bell System Technical Journal vol. 27, no. 3, (1948) pp 379–423

Smith, G. and Whitehead, J. Analyzing the Expressive Range of a Level Generator. In Proceedings of the 2010 Workshop on Procedural Content Generation in Games.

Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley and Cameron Browne (2011): Search-based Procedural Content Generation: A Taxonomy and Survey. IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG), volume 3 issue 3, 172-186.