Getting Started With Danesh

Setting Up Your Generator

To add a new procedural generator to Danesh so you can start exploring it, we first need to tell Danesh a few things:

- How to get a new piece of content from your generator
- What parameters or variables we want Danesh to be able to change
- How to display a piece of content on the screen

For this guide, we'll be using the Cellular Automata Cave Generator example generator, which is included in the files for Danesh.

Annotating The Generator

Danesh uses a C# feature called *annotations* to get more information about your generator. Annotations are lines of code that appear above methods or variables. The first annotation we're going to add is the one that marks the main generator method - the bit of code that produces content. This method should have **zero arguments**, and **return an object** type.

To annotate the method, we write `[Generator]` above the method name, like this:

```
[Generator]
public object GenerateCave(){
      //. . .
```

That's it! That lets Danesh know that it should call this method whenever it wants to generate content. If you don't have a convenient method like this you can always write one just for Danesh to use, that sets up your generator and calls whatever methods it needs to.

Annotating Parameters

Parameters are the levers and switches we can use to change how our procedural generator works. A parameter is any variable that affects how the procedural generator behaves. In a dungeon generator it might be the chance of adding treasure to a room, or in a tree generator it might be the range of colours leaves can be. Annotating variables like these lets Danesh find them and connect them to the interface for us to experiment with.

This annotation is different to the `[Generator]` annotation, because it has extra information in it. Here's an example from the Cave Generator:

```
[TunableParameter(MinValue: 0, MaxValue: 8, Name: "No. Of Iterations")]
public int numIters = 5;
```

This variable, which affects how many times the generator runs a particular bit of code, uses the `TunableParameter` annotation, which has three extra pieces of information: `MinValue` and `MaxValue`, which tell Danesh the limits on this variable, and `Name`, which offers a plain-language description of what the variable does (this makes it easier to remember what a parameter is when using Danesh).

You can change these easily later, so don't worry if you don't know exactly what values to use. Experimenting and trying things out is a big part of using Danesh! Good values to start with are the legal limits for a variable - for example, many parameters will not be able to be negative, so zero can be a good starting minimum value.

Writing A Visualiser

In order to show you content that's been generated, we need to give Danesh a way to represent it visually. In the current version of Danesh you do this by writing a *visualiser*, a method that draws onto a texture which we display on the screen. In CellularAutomataGenerator there are two example visualisers - here we'll step through the simpler one, RenderSimpleMap, line-by-line:

```
[Visualiser]
public Texture2D RenderSimpleMap(object _m){
```

Visualisers must all have the same method signature: they must return a Texture2D, and take an object as its sole argument. The object passed to this method is the content generated by your generator method that we annotated earlier.

```
      Tile[,] map = (Tile[,]) _m;
```

The first thing we do is cast the object to the type we expect. If you get an unexpected piece of content, from another generator for example, this will fail but Danesh won't crash.

```
      Texture2D tex = new Texture2D (
            map.GetLength(0),
            map.GetLength(1),
            TextureFormat.ARGB32, false);
```

The above line creates a new texture for us to draw our content onto. In many cases, we may want a specific size texture to make drawing easier (in this case, we're making a new texture that is exactly the width and height of our map.

```
for(int i=0; i<map.GetLength(0); i++){
      for(int j=0; j<map.GetLength(1); j++){
```

In this example visualiser our content is a two-dimensional array of tiles that can be either solid or empty. We're using two for loops to iterate over the width and then the height of the map. Then, for each tile on the map, we draw a coloured pixel to the texture to indicate whether it's solid or empty:

```
if(map[i,j].BLOCKS_MOVEMENT)
      VisUtils.PaintPoint(tex, i, j, 1, Color.black);
else
      VisUtils.PaintPoint(tex, i, j, 1, Color.white);
```

VisUtils is a helpful class with methods in to assist you in writing visualisers. PaintPoint takes a texture, an x and y co-ordinate, a scale factor and a colour, and paints a region of the texture. We use it here to paint one pixel at a time.

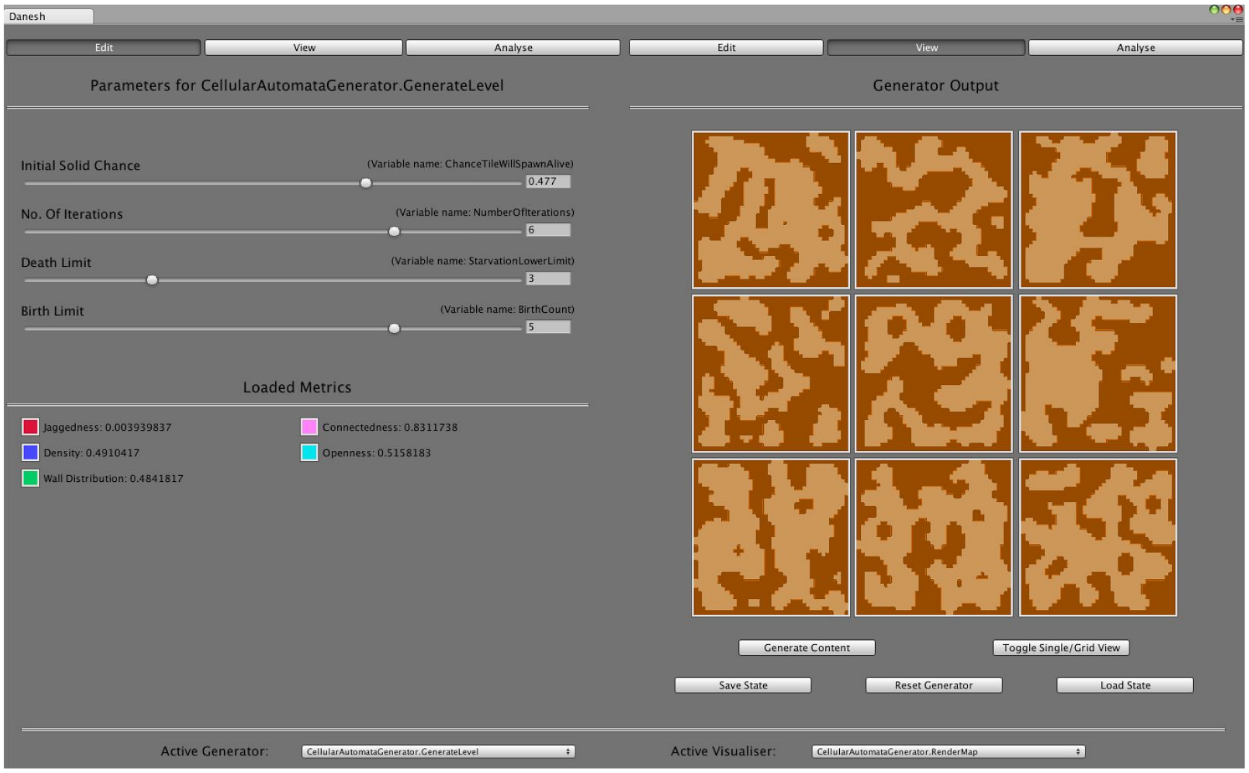
```
tex.Apply();
return tex;
```

Before returning the texture to Danesh, it's important we call Apply on the texture to finalise our changes. Then we return the finished texture to Danesh, which it displays on the screen.

Exploring Danesh's Interface

Opening Danesh

Danesh has its own window in Unity. You can open it by clicking the Window option at the top of the screen, and selecting Danesh. The main window should look something like this when it opens, if it's successfully managed to find and load your generator:



Parameters for CellularAutomataGenerator.GenerateLevel

Initial Solid Chance (Variable name: ChanceTileWillSpawnAlive) 0.477

No. Of Iterations (Variable name: NumberOfIterations) 6

Death Limit (Variable name: StarvationLowerLimit) 3

Birth Limit (Variable name: BirthCount) 5

Loaded Metrics

🟥 Jaggedness: 0.003939837
🟪 Connectedness: 0.8311738
🟦 Density: 0.4910417
🟦 Openness: 0.5158183
🟩 Wall Distribution: 0.4841817

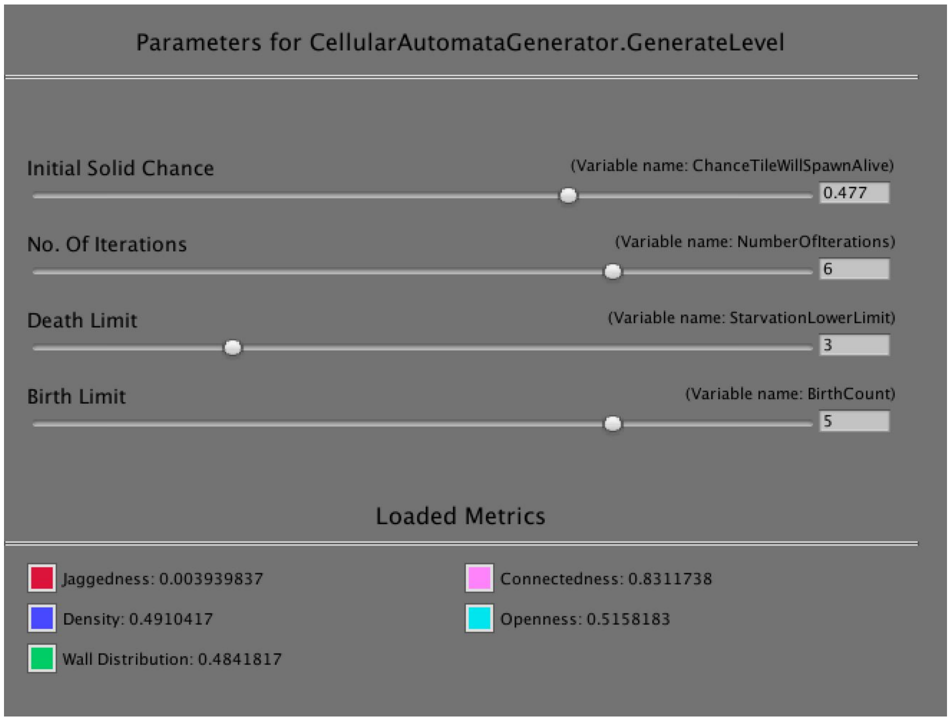
Active Generator: CellularAutomataGenerator.GenerateLevel

Active Visualiser: CellularAutomataGenerator.RenderMap

Fig. 1: The Danesh interface, showing the Edit tab (left) and the View tab (right).

Danesh is spit into two panels, one on the left and one on the right. Each panel can display one of Danesh's task windows by clicking the tabs at the top of the panel. Right now there are three tabs: **Edit**, **View** and **Analyse**. In this section we'll step through each tab and explain what they do.

If you loaded multiple generators, you can select the generator you wish to work on using a drop-down menu at the very bottom of the window. Danesh remembers the state of each generator, so you won't lose your work on one generator if you go and work on another.


Parameters for CellularAutomataGenerator.GenerateLevel

Initial Solid Chance (Variable name: ChanceTileWillSpawnAlive) 0.477

No. Of Iterations (Variable name: NumberOfIterations) 6

Death Limit (Variable name: StarvationLowerLimit) 3

Birth Limit (Variable name: BirthCount) 5

Loaded Metrics

🟥 Jaggedness: 0.003939837
🟪 Connectedness: 0.8311738
🟦 Density: 0.4910417
🟦 Openness: 0.5158183
🟩 Wall Distribution: 0.4841817

Fig. 2: The Edit Panel.

The Edit Panel

The Edit panel displays all the parameters annotated with the `TunableParameter` annotation. For each parameter you'll also see a control to change its value - currently that means a slider for numeric values, and a checkbox for boolean values. Any change you make using the controls immediately affects the generator. Currently you cannot edit the text boxes to directly set the value - see *Future Updates* for more information.

Beneath the parameters you'll see a list of metrics. If you wrote any metrics, Danesh will check to see which ones work on the currently active generator. If a metric works, it'll be displayed here. Metrics are also assigned a colour, which is used to quickly identify them in visualisations later. Colour customisation is planned for a future update.

Next to each metric is a number. This represents either the metric score of the last thing Danesh generated or, if multiple pieces of content are on-screen, the *average* metric score for all the pieces of content being displayed. In Figure 1 on the previous page there are nine pieces of content showing in the right-hand tab, so the metric scores are an average of all nine results.

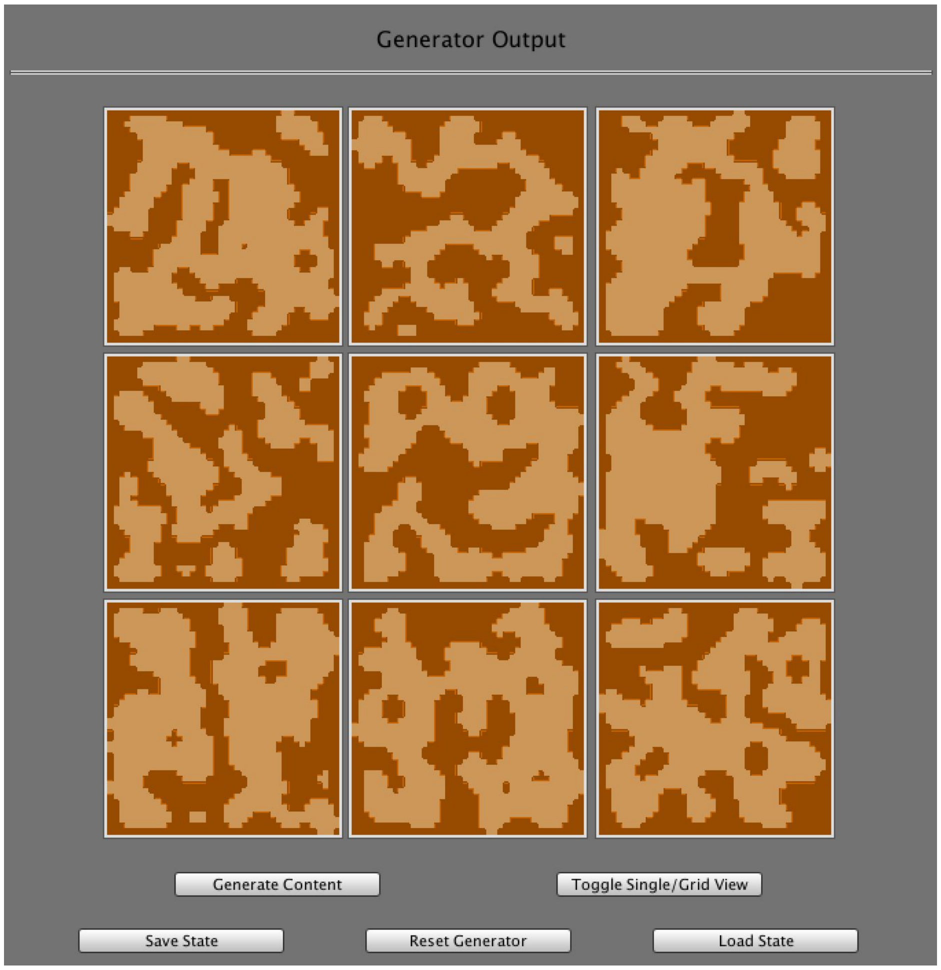


Fig. 3: The View Panel.

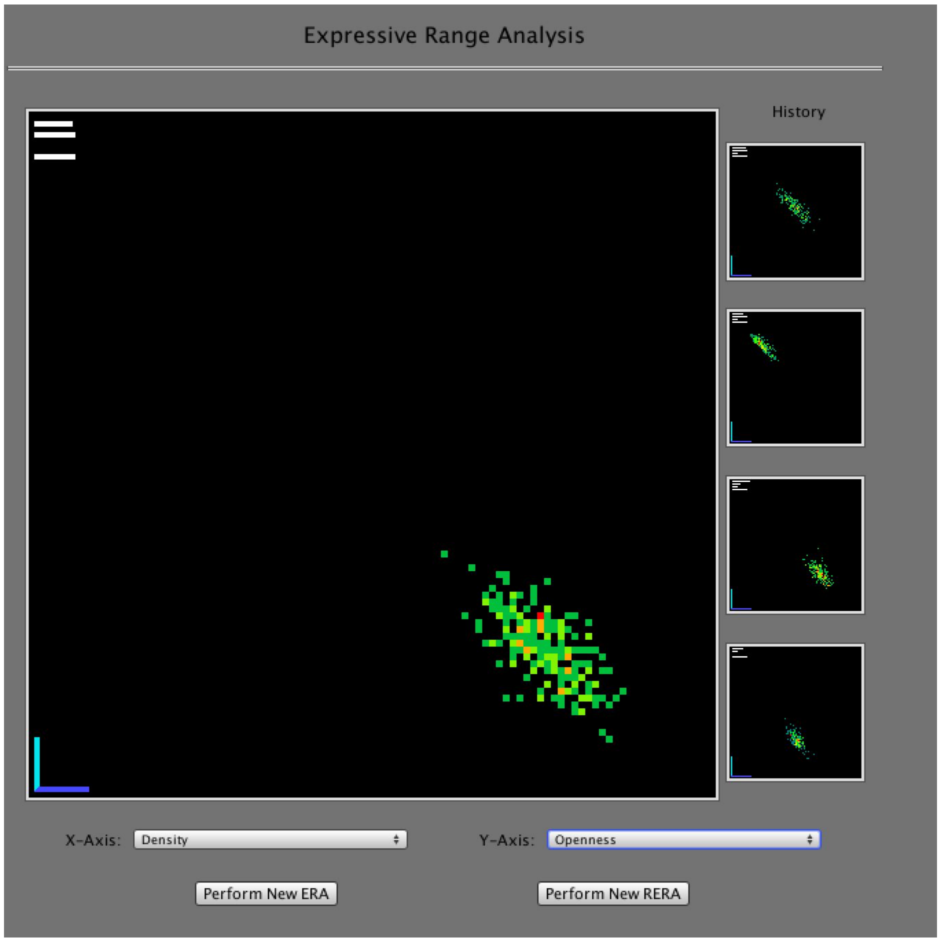
The View Panel

The View panel displays output from the generator, using the visualisers you wrote for it. If you have multiple visualisers you can select a different one using the drop-down menu at the bottom of the Danesh window, next to the generator selection. To generate new content, press the ‘Generate Content’ button.

In the main part of the window some output from your generator is displayed. You can switch between showing one or nine pieces of content at once using the ‘Toggle Single/Grid View’ button. Note that showing nine pieces of content currently takes longer to render - if your generator is slow, we recommend using the single-output view.

Beneath the generation buttons are three other buttons for saving, loading and resetting the generator. Saving a generator stores all of its current parameter settings in a file within your Unity project. When you click ‘Save State’ you can give the save state a name before saving. To load these save states, simply press ‘Load State’ and Danesh will display a list of save files it can see. Note that currently this process is not safe, meaning Danesh will show save files for other generators as well, and if you have modified the generator since creating the save state (e.g. removing or renaming a parameter) this may result in unexpected behaviour.

The ‘Reset Generator’ button restores all the parameters back to the values they had when you opened the Danesh window. This process can’t be undone. Note that if you close Danesh and immediately reopen it, it will store a new set of parameter values to reset to. This means that if Unity crashes or has to be restarted for any reason you may lose progress or lose the original state your generator was in, if you were experimenting with it. For this reason, we recommend users **save early** and **save often**.



X-Axis: Density
Y-Axis: Openness

Fig. 4: The Analyse Panel.

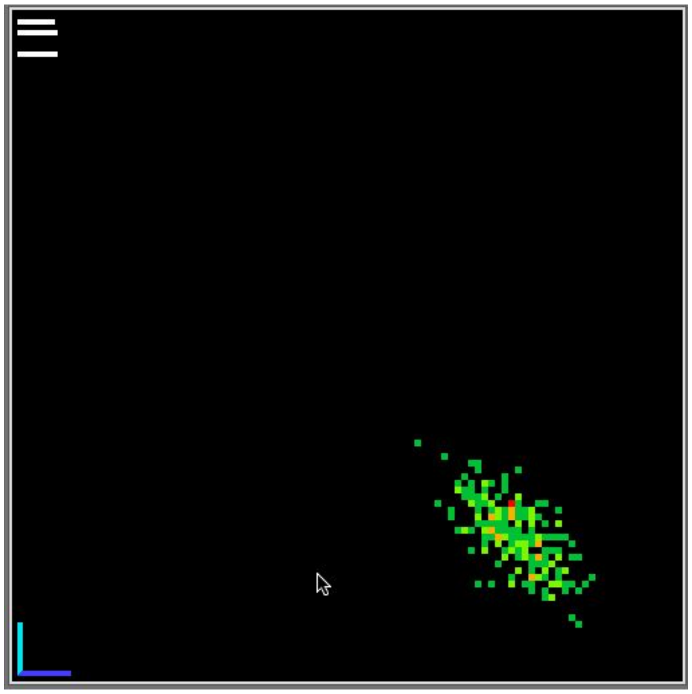
The Analyse Panel

The Analyse panel lets you create visualisations that show the shape and behaviour of your generator in an abstract, high-level way. This section will go over the basics of generator analysis, but we dive into more detail in the *Case Studies* section of the manual, later on.

Expressive Range Analyses

The 'Perform New ERA' button performs an *expressive range analysis* on your procedural generator. What this effectively means is Danesh samples your generator hundreds of times, and each time makes a note of the metric scores for each sample. It then represents all of these samples visually on a histogram. The axes for the histogram are two metrics, which you can change around. Just underneath the ERA are two drop-down menus, one for the x-axis and one for the y-axis. Click on them to change the metrics being displayed.

In the above example, a heatmap for the **density** and **openness** metrics is being shown. The ERA is clustered in the bottom-right quarter of the histogram, meaning the content has a high density score (which is represented on the x-axis) and a low openness score (represented on the y-axis). Green areas indicate small quantities of samples, through to red areas which indicate very high concentrations of samples.



To see what a particular point represents, we can hover our mouse over it and Danesh will show us an example piece of content that has these exact metric values. This is a real sample collected during the ERA.

Performing Multiple ERAs

If you change your generator the ERA won't change, because it only shows data it has already collected. This means you must run a new ERA every time you want to see the effect of your changes on the generator. Each time you run an ERA, the old ones are added to a history of ERAs on the right-hand side. This is to help you compare recent changes to the generator by performing an ERA after each change. We'll touch more on this later on in the manual.

Performing Random ERAs
A normal ERA shows you a picture of what your generator is currently producing. Danesh can also try and give you a picture of the full potential of your generator by performing a *Random ERA*. When you click the 'Perform New RERA' button Danesh will sample your generator as before, but this time it will take many more samples. Each time it samples the generator it will randomly change the parameters of the generator. This way, each sample is a snapshot of a different configuration your generator could be in.

RERAs appear slightly differently to ERAs. We don't apply heatmap colourings as they are typically far sparser than standard ERAs. We also don't show a history, since changing the generator doesn't affect the outcome of the RERA (you can of course run a RERA multiple times, to get new samples).

Just like ERAs, you can hover over an RERA data point to see an example piece of content with those values. Each content example comes from one of the randomly-configured generators that Danesh tried while making the RERA.

If you compute an ERA and an RERA, you'll see an option to switch view modes between ERA and RERA. Danesh keeps track of which metrics you were looking at for each one.

Case Studies: How To Use Danesh

We've introduced you to all of Danesh's features, but in order to show you how to get the most out of the tool we've prepared a few example problems. You can follow along with these examples by loading up Danesh with the generator we mention in each case study. You can also just read the case studies to get a feel for the kind of problems Danesh can help with.

Making Danesh easy to use and understand is really important to us. If you have anything you'd like to see added here, or think something isn't clear, please let us know: danesh@metamakersinstitute.com.

Case Study: Disconnected Dungeons

Say we've written a cool cave generator to use in our RPG, but there's a problem: sometimes the player can't reach the exit, because the cave isn't 100% connected. We've tried fiddling with parameters but we can't get a good result. Maybe Danesh can help? In this case study, we'll look at the Cellular Automata Generator and go through the stages of identifying a problem, describing it using metrics, running ERAs and finding a solution.

Step One: Describe The Problem

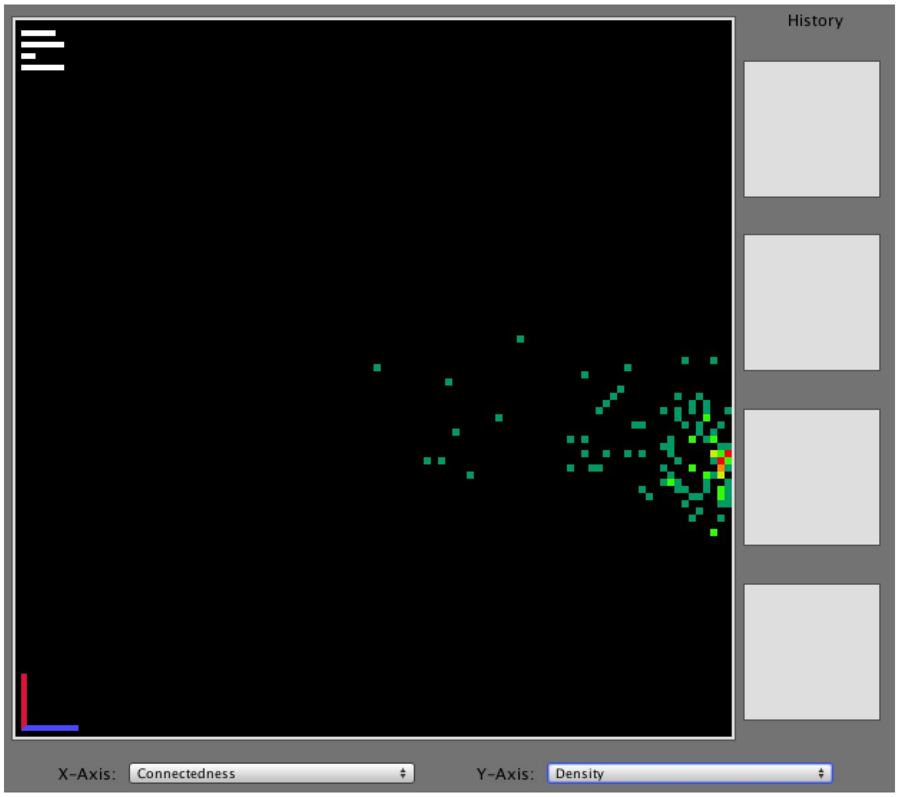
Deciding what metrics to write can be tricky. If you're just exploring your generator you can write a metric for anything you think is interesting - ideas about aesthetics, or difficulty, or variety or complexity. Sometimes you have a very specific problem in mind though. In this case study our problem is to do with accessibility - what bits of our generated dungeons are connected to each other?

We're going to write two metrics to try and help understand this. You can find both of the metrics in the file LevelAnalyser.cs. The first is `CalculateDensity`, which returns a number between 0 and 1 that represents what percentage of the map is solid wall tiles. Too many solid wall tiles is likely to cause disconnected sections of our dungeon, so this metric will let us see which parameters affect that, and search for levels that aren't too dense.

The second metric is `CalculateConnectedness`. This metric returns a number between 0 and 1 representing the percentage of empty tiles that are in the biggest contiguous group of empty tiles. So if the level is one big empty space, connectedness is 100%. If the level is made up of two disconnected empty areas that are exactly the same size as one another, the connectedness is 50%, and so on. We want our generator to produce levels with the highest possible connectedness - ideally 100%. Writing metrics that describe a property we either really want, or really want to avoid, helps us focus in on important things with Danesh.

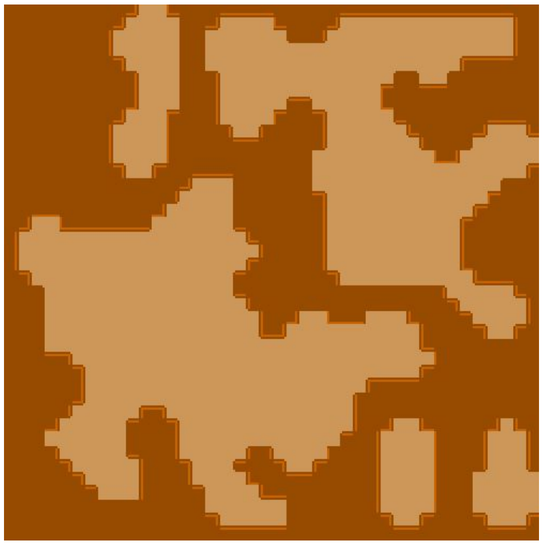
Step Two: Perform An ERA

Now that we have some metrics to describe content, we can perform an Expressive Range Analysis, or ERA, to look at our generator and see what metric scores our content currently has. To perform an ERA, click on the Analyse tab on either side of the screen, and then click the Perform ERA button. A progress bar will fill up and when it's complete you'll see an ERA appear in the panel. An example ERA is shown below.



When Danesh performs an ERA it generates hundreds of pieces of content using your generator, measures the metric scores for each piece of content, and then plots the resulting data on a histogram. The axes for the histogram are the metrics we've given to Danesh. In the example above, the x-axis represents `Connectedness` scores, and the y-axis represents `Density` scores. Each dot on the histogram represents one or more pieces of content generated by Danesh with those metric scores. The colour scheme is like a heatmap: green means just a few datapoints, while red means a large concentration of data.

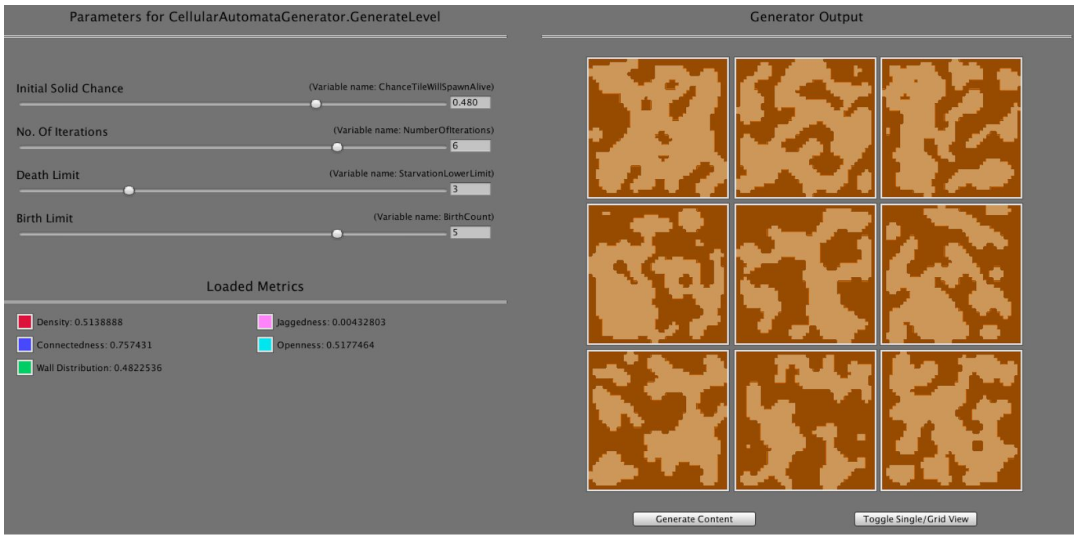
In the ERA above we can see that although most of the data points are on the far right of the Connectedness axis, some of the points are much lower, around 70% or even 50% connectedness. What do those points look like? We can hover our mouse over them to see what data they represent. Danesh will pop up an example piece of content with those metric values. Here's one of the outliers with approximately 50% Connectedness:



A perfect example of our problem - disconnected dungeons! Now to try and fix it.

Step Three: Make A Change

Now we have a good set of metrics for describing our problems, we can begin to change our generator and see happens. On the left-hand panel, click on the Edit tab. On the right-hand panel, click on the View tab. Now we can change parameters and click ‘Generate Content’ to see the result. You can click the “Toggle Single/Grid View” button to view either one or nine pieces of content at once. Nine pieces takes longer to generate, but it makes it easier to see at a glance what your generator is doing.



Parameters for CellularAutomataGenerator.GenerateLevel

Initial Solid Chance (Variable name: ChanceTileWillSpawnAlive) 0.480

No. Of Iterations (Variable name: NumberOfIterations) 6

Death Limit (Variable name: StarvationLowerLimit) 3

Birth Limit (Variable name: BirthCount) 5

Loaded Metrics

🟥 Density: 0.5138888
🟦 Connectedness: 0.757431
🟩 Wall Distribution: 0.4822536
🟪 Jaggedness: 0.00432803
🟦 Openness: 0.5177464

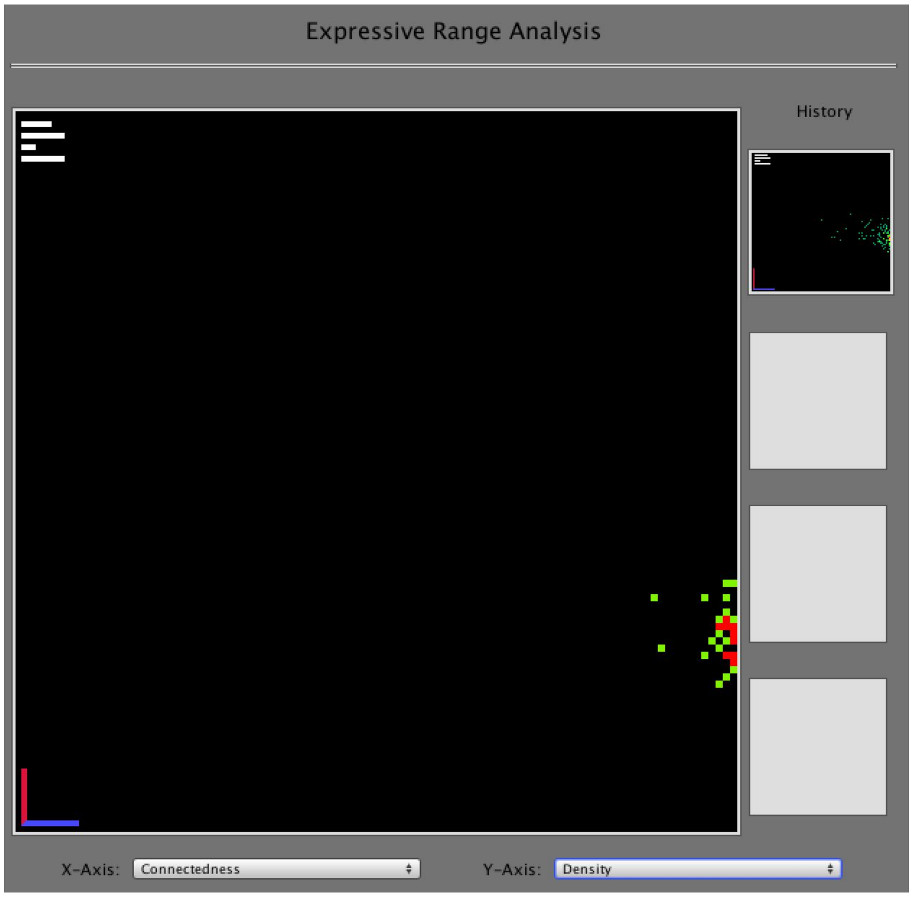
In the Edit tab we can drag sliders to change parameters of the generator, and then click Generate Content to see new output generated with these values. Try changing some parameters and generating content. As you generate content, you’ll see the numbers in the

‘Loaded Metrics’ section update. These numbers are the scores for the generated content. This is useful in case the thing you’re looking for can’t easily be seen with a quick visual glance.

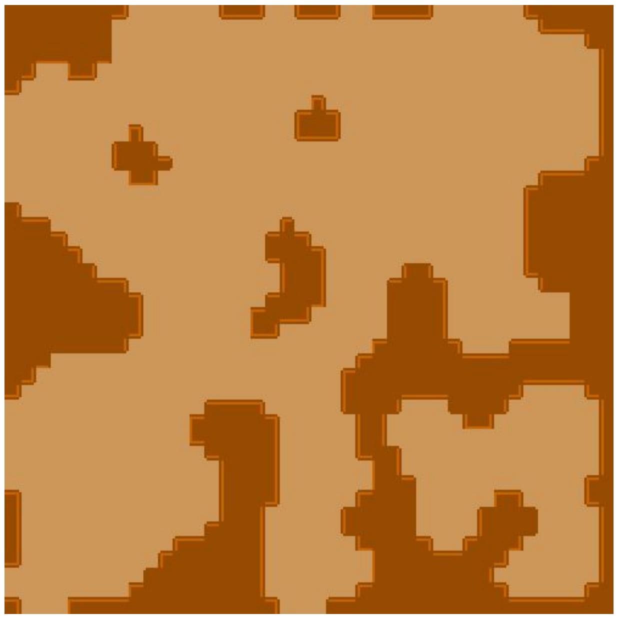
As we experiment with the sliders, we’ll start to notice that some parameters have certain effects on the output. For example, increasing the ‘Initial Solid Chance’ parameter will make the dungeons denser, and lowering the ‘No. Of Iterations’ parameter will make the dungeons more jagged and scattered.

If we forget where we started from, we can reset our generator back to how it was when we loaded Danesh. Underneath the generated content, click the ‘Reset Generator’ button, and then click ‘Yes’. Alternatively, if we find a generator we want to keep for later, we can use the Save/Load functions to save a particular set of parameters.

Step Four: Confirm Your Results
Once we find some new values that we think have solved our problem, we can confirm this by running another Expressive Range Analysis (remember, click on the Analyse tab, and then click ‘Perform ERA’). Just like before, this will sample and measure hundreds of pieces of content and then represent them in a histogram. What we hope to see is that the data points with 50% or 70% Connectedness have disappeared, and instead most of our data is to the right-hand side.



Above is an example of an ERA after changes have been made - you can see this particular one by loading the DisconnectedDungeons save state that we've included with Danesh. Note how all of the data points are clustered to the far right, indicating they all have 90%+ Connectedness. You can scroll back in this document to see the old ERA, but if you're working in Danesh it lists the last four ERAs you generated in the History column on the right-hand side. You can see in the above image that our last ERA is also displayed there, letting us quickly compare the two ERAs. We can easily see that we've managed to reduce outliers, and even hovering our mouse over the least connected data point, we can see a great improvement:



In a future Danesh update we'll be adding Automatic Optimisation to help you get through this process with Danesh's help. For more information, see the **Future Versions** section of this manual, further below.

Case Study: Untapped Potential

We've found the source code for an interesting procedural generator online, but the parameters are confusing and we're finding it hard to make changes that don't completely break it. What kind of things can we achieve with our generator? In this case study we use Randomised Expressive Range Analysis, or RERA, to explore thousands of parameter combinations and find interesting results that we like the look of. This case study uses the Two Phase Cellular Automata Generator.

Step One: Describe Interesting Features

In the 'Disconnected Dungeons' case study we were using Danesh to solve a problem, so we wrote metrics that described features we thought might be related to the problem. When we're exploring a generator freely, we might not know exactly what we want to look for. What kind of metrics should we write? There are a few different approaches that we've found - let us know if you have any other ideas!

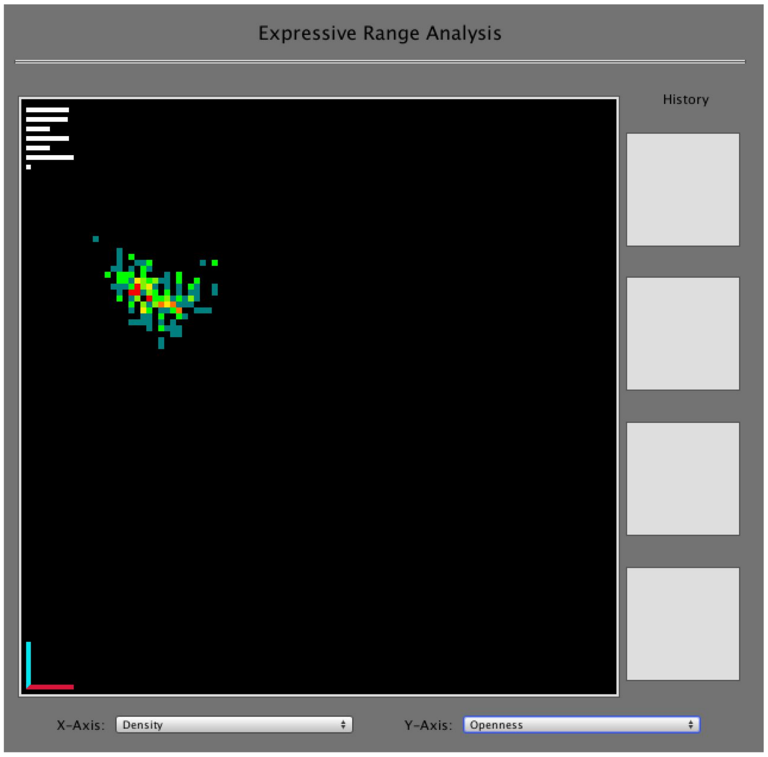
One is to identify features that you're interested in seeing in your generated content. If we're generating dungeons and you want lots of corridors and passages, you might write a metric that calculates how many of the empty tiles are adjacent to a wall. When that number is very high, the maps are mostly tight corridors. When that number is low, the empty space is all clumped together in big open rooms. A metric like that can help you look for a particular feature in the RERA and ERA histograms.

Another approach is to write metrics that differentiate the content in your generator. By this, we mean a metric that you think the generator is able to score both very low and very high in. For example, if we are making a maze generator and we measure what percentage of tiles are dead ends (that is, a square surrounded by three walls) this metric will almost always score very low, because most tiles in a maze are corridors that lead to either branches, dead ends or exits. But if we write a metric that measures what percentage of tiles are on paths *which lead only to dead ends*, we can imagine mazes that are very open with paths that link back to themselves and few dead ends, as well as mazes with lots of branches that lead to nothing.

If you're exploring a generator, writing any metric you can think of will give you some insight into what's going on, so don't be afraid to explore. Think about something you can measure about your content, then write a bit of code to do it. The first metrics you write might reveal something to you that inspires you to come back and write more. If you need inspiration, for this case study you can use the metrics found in LevelAnalyser.cs.

Step Two: Perform an ERA

Before we look at the whole possibility space of our generator, it can be helpful to see what our generator is currently doing, in order to understand what kind of metric scores the content it currently generates has. To do this, click on the Analyse tab in Danesh, and then click ‘Perform New ERA’. A progress bar will fill up while Danesh samples your generator, and then a histogram should display. An example is shown below:

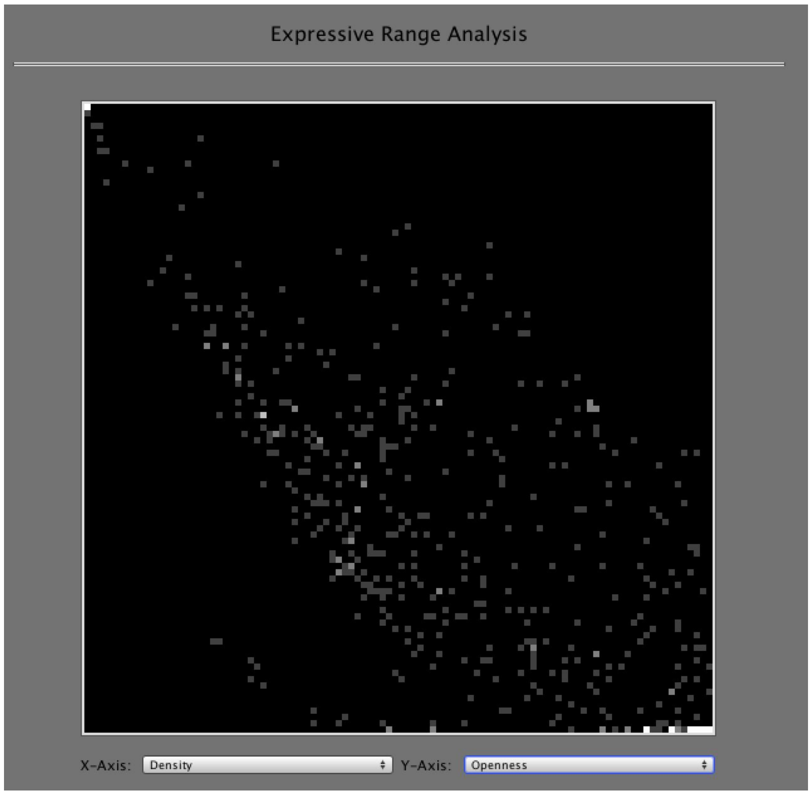


When Danesh performs an ERA it generates hundreds of pieces of content using your generator, measures the metric scores for each piece of content, and then plots the resulting data on a histogram. The axes for the histogram are the metrics we’ve given to Danesh. In the example above, the x-axis represents `Density` scores, and the y-axis represents `Openness` scores. Each dot on the histogram represents one or more pieces of content generated by Danesh with those metric scores. The colour scheme is like a heatmap: green means just a few datapoints, while red means a large concentration of data.

Underneath the ERA are two drop-down menus that control which metrics are shown on the histogram. We can select different metrics and see where our generator sits in all of them. This will give us a good idea of different areas that we might want to explore with an RERA. For example, in the screenshot above we can see that this generator currently has low average density and relatively high average openness. We may wish to explore other quadrants of the histogram with our RERA.

Step Three: Perform an RERA
Our ERA showed us the kind of content our generator outputs in its current state. But we don’t really know what the parameters control in this generator, so it won’t help us much to change parameters and look at more ERAs. What we need to do is to look at lots of different options

quickly, and pick out interesting points. To do this, we're going to perform an RERA, by clicking on the 'Analyse' tab and clicking the 'Perform New RERA' button. This takes longer to compute than an ERA, but when it's done a new histogram (in black and white) should appear. An example RERA is shown below:



When Danesh performs an RERA it generates thousands of pieces of content using your generator, like a larger-scale ERA. Unlike an ERA, however, each time it generates a piece of content it randomly sets every parameter to a new value, using the minimum and maximum values you defined as a guide. This means that each of the thousands of pieces of content comes from a different parameterisation of your generator, and it's this that will allow us to use the RERA to see the breadth and diversity of our generator's content. As with the ERA, Danesh calculates metric scores for each of these pieces of content, and plots the result in a histogram.

Just like an ERA, we can mouse over a data point to see an example from a data point, and change the metrics we're viewing by selecting options from the drop-down menus beneath the RERA. Take a few minutes to mouse over some data points and see the variety of samples found by Danesh.

RERAs are useful for all sorts of things. One thing they can do is tell us things that our generator doesn't seem to be able to do. For example, in the RERA shown above, there is no data in the top-right hand corner of the histogram, which means that out of all the thousands of random parameters settings Danesh tried, it never found a piece of content with very high `Openness` *and* very high `Density`. This might help us learn something about the limitations of this generator we're looking at.

Another thing we can do is instantly setup our generator to match an example piece of content that we like. Hover your mouse over some of the histogram points until you find an example you like the look of, then left-click it. On the other side of the screen, click the 'View' tab and you should see that the generator is now producing content similar to the example you clicked on. Danesh records the parameter settings for every RERA data point, and can instantly load them into the generator when you ask. Click on a few more points to see it generate different results.

Step Four: Save Interesting Points In Space
Once we've found something we like the look of in the RERA, we can examine it in more depth by left-clicking it. Danesh will change the parameters on our generator to match the parameters that produced the example we saw. We can now click on the View tab and click Generate Content to see more examples from this generator, or go to the Edit tab to see what parameters this data point had.

To avoid losing this result, we might want to save this generator so we can go back to it later. Click on the View tab, and at the bottom click on the Save State button. This feature lets us save the parameters of this generator so Danesh can load it again later. Type in a name for this generator - or keep the random one Danesh gives you - and click Save. You can load this again later by selecting this generator, going to the View tab, and clicking Load State.

Now that we've saved this state, we can safely go back and look at another one, change the parameters to tweak it a little bit, or run an ERA to learn more about this new configuration!

Coming In Future Versions!

Danesh is built on top of cutting edge research, and we have lots of plans for new features and ideas we want to integrate into the tool. Below are a list of some plans we have, and an indication of how close we are to adding them in. If you have any requests or suggestions, please email them in to us! You can get in touch with us at danesh@metamakersinstitute.com.

Automatic Optimisation

The most common way to work with procedural generation, and the way Danesh currently works, is to change parameters and then look at the output to see if it matches what you want. But with Danesh's Metrics, we can invert this process and let you describe the output you want (in terms of metric values) and ask Danesh to go and look for parameters that produce that result.

We've already implemented automatic optimisation, and it's a really great way of solving problems with procedural generation. We left it out of 1.0 because we wanted to polish up the other features first, but expect this to come in an update soon! We wrote a paper about some of our techniques here: http://research.gold.ac.uk/18950/1/Cook_CIG2016.pdf

Parameter Analysis

One of our objectives with Danesh is to give you as many ways as possible to think about your procedural generator and interact with it. We're working on some analytical techniques that will let Danesh tell you things about your parameters, like how unpredictable they are, what other parameters they're connected to or influenced by, and how much they impact certain metrics.

This analysis involves some pretty funky maths and is firmly in the experimental phase. You can always check up with us on Twitter for updates and news, though: @thosemetamakers for general updates and @mtrc for updates from Mike, one of the developers on Danesh.

Tracery + Other Grammar Tools

Tracery is a super-cool and popular way to write grammars, especially for generating text. Grammars are used all over procedural generation and we're planning a suite of tools designed especially for analysing grammars and helping people get more information about their strengths and weaknesses.

This will probably be our priority after adding in automatic optimisation. It'll involve some experimental interface work, and being general across lots of different ways to represent grammars might be tricky, so we don't have a firm timeline, but we're hoping we can add some really useful features that will help people with this popular technique.