

# Seeds



*Issue 1*





# Credits

**Editor:** Jupiter Hadley - @Jupiter\_Hadley

**Production Assistant:** Chris Bowes - @contralogic

## Contributors:

Jonathan Pagnutti

Gillian Smith

Tommy Thompson

Jonas Delleske

Balint Mark

Joao Oliveira

Marek Skudelny

Lena Werthmann

@TearOfTheStar

Oliver Carson

Luke O'Connor

Todd Furmanski

Sam Geen

Isaac Karth

David Murphy

Ciro Duran

David Morrison

Eggy Interactive

Davide Aversa

Kevin Chapelier

Ahmed Khalifa

Max Kreminski

Gabriella A. B. Barros

Ahmed Khalifa

Tim Stoddard

Martin Černý

Jo Mazeika

Kate Rose Pipkin

Aidan Dodds

Ex Utumno

Afshin Mobramaein

Scott Redrup

Mark Johnson

Kate Compton

Johanthan Pagnutti

Jim Whitehead

Mark Bennett

Niall Moody

Katie Compton

Gregoir Duchemin

Dave Griffiths

Rune Skovbo Johansen

## Organisers:

Michael Cook, Azalea Raad

## Press Officer:

Jupiter Hadley

## Art By:

Khalkeus, Tess Young

## Speakers:

Gabriella Barros, Joris Dormans,

Becky Lavender, Mark Nelson,

Emily Short, Tanya Short, Adam

Summerville, Jamie Woodcock

**Thanks to:** Heidi Ball, Simon Colton, Mark Nelson, Blanca Pérez Ferrer, The Metamakers Institute, Falmouth University, Sekrit Inc. and everyone in the PROCJAM community

**Cover art by:** Kevin Chapelier

Some header/Footer patterns from Eduardo Lopes' Procedural Fabrics Generator:

<https://eduardo.itch.io/procedural-fabrics>

# ProcJam

*Make Something That Makes Something*

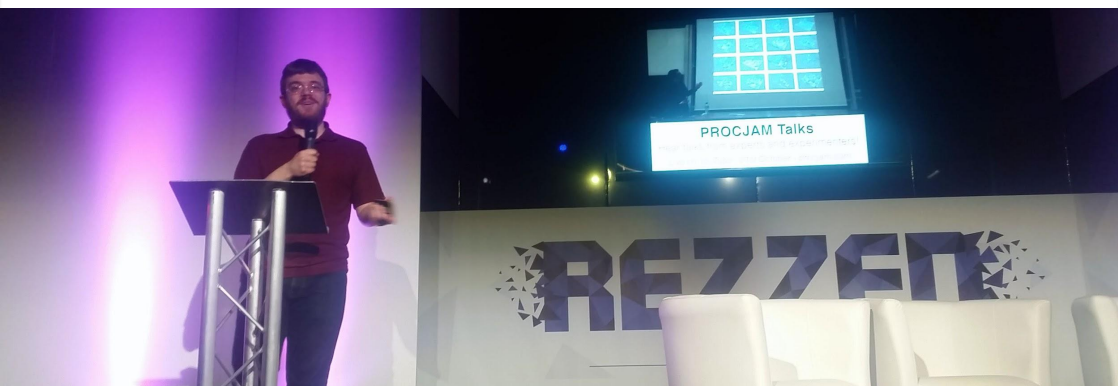
The ProcJam is not like most other game jams. This jam is aimed at making procedural generation accessible to more people and to show off projects that are pushing the boundaries of generative software. This jam is easy to enter, laidback, and fun to be apart of. We are building a community of friends and peers across disciplines all interested in procedural generation.

This game jam takes place across nine days, including two weekends. You can enter anything you'd like - art, boardgames, tools, games, anything you can think of, as long as it has something to do with procedural generation/random generation/generative software, ect. You can even take an existing thing and add some generative magic to it for the jam! If you start before the jam or want to finish the jam later, that is fine too.

We have a kickoff day at the start of the jam, taking place in Falmouth this year, where loads of awesome speakers are going to talk about procedural generation. This unconference is livestreamed that day, as well as put up online to be watched in the future.

The ProcJam is happening as a part of Metamakers Institute's 'Games as Arts/Arts as Games' festival.

This Zine was made by the ProcJam community. We hope you enjoy it!



# How is Music Like A Spring?

By Jonathan Pagnutti

Music is weird.

Once upon a time, I did a lot of music theory. I actually ended up with a music performance minor rather than major because I took two extra classes of music theory rather than the required music history classes of my undergrad (nerd alert). My final project as an undergrad was a music generator, which I wrote in C and only exists on a rapidly aging desktop computer collecting dust.

One of the wonderful things about procedural generation is that if we can encode a theory into the computer, we can have a computer generate endless examples of stuff that follows the theory. Even better, if we line up our metaphors, the magic box can present something in terms of something else.

So, the question at hand: how is music like a spring?

When you push or pull on a spring, you add tension to it. Let go, and the spring releases that tension. I'm sure this tension has a special name, but I only ever took freshman physics. Music carries and releases tension too, according to a music theory called Functional Harmony. But, first, we need to take a whirlwind tour on sheet music.

Note names on the musical staff. These form a musical alphabet from A-G.

A scale is a collection of notes that goes through the musical alphabet. This is the C scale, because it starts on C and repeats on every C

I ii iii IV V vi vii(dim) I

do re mi fa so la ti do

Think of *Do Re Mi* from the *Sound of Music*. That song is a scale!

We can build groups of notes that play at the same time from each note in the scale. These chords have special labels that tell us what scale letter is at the bottom. These labels can also tell us which chords pull us towards other chords.

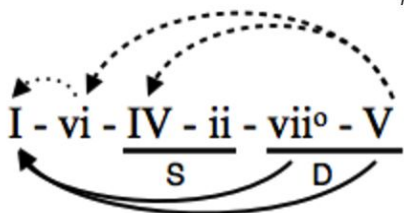
Ok, so collections of notes played at the same time (called chords) can get special labels based on their lowest note in a scale. The cool thing is that those special labels can tell us which chords carry lots of tension they want to release, and which chords are less stressful.

They can also tell us how to go from chord to chord to increase and decrease tension.

Decreasing Tension ←

*Functional Harmony* is a theory on which chords should follow which.

You can always increase tension by moving to the right (even being allowed to skip chords)  
But to go leftward, you need to follow the arrows.



Dotted arrows point to places you don't want to stay for very long.

→ Increasing Tension

Fig. 2

This chart was ruthlessly stolen from Dmitri Tymoczko's 'A Geometry Of Music'. Annotations are mine. The degree symbol by the vii is the same as the (dim) by the vii in the earlier picture.

So, then, we can push and pull our spring and play chords at various tension levels. In theory, as the spring expands and contracts, we'll get 'pleasing' chord progressions. There is a bunch more to consider to turn these roman numerals into sound, but the raw idea comes from this theory. I encoded this theory, and using P5.js made a tiny little generative audio prototype.



Music generation prototypes don't make the best screenshots.

You can try it out for yourself at

[http://www.tinyai.net/projects/musical\\_springs](http://www.tinyai.net/projects/musical_springs)

Now, why make this? Music, even generative music, seems to be tethered to events or notes, but it doesn't have to be. Music is just as pervasive and continuous as, well, physics. I grounded this tension-release model in Functional Harmony because I know it, but music generates tension in so many other ways. With this tension-release model, we're starting to get at musical velocity. And if that has a nice metaphor, maybe we can describe a collision musically?

Music is more than the notes on the page. Music is weird.



# Why Do We PCG?

By Gillian Smith

*"We are drawn to the promise of infinite (or at least, really huge) amounts of content to explore."*

I want to talk about "replayability" and meaning and depth. We often talk about how PCG can give people different experiences each time they play. We are drawn to the promise of infinite (or at least, really huge) amounts of content to explore. And then we, inevitably, are disappointed by the infinite: there's a lot of it, but it all feels so similar. It is unrealistic to hope for constant, infinite beauty.

Why do we replay games? Why do we re-read books, or re-watch movies, or re-listen to music?

We don't hope for books to be infinitely long. Sometimes we want to hear more about the characters after reading a favorite novel, but ultimately it's probably for the best that we don't. It's better to yearn for and imagine what happens next. Stories that have potential futures have a power to them. Would an infinitely long story, where our urge to know what happens next is always fulfilled, be enjoyable to read forever?

We don't need a beloved movie to have different content every time we watch it in order to find it fulfilling. With some movies, there is satisfaction in finding things we missed earlier viewings: elements of foreshadowing, interesting background character behavior, and clever cinematic tricks. Sometimes we find our experience of watching the movie has changed, because though the movie's content has remained static, we have changed as viewers and are reacting differently to the same material.

We don't ask composers to write music that adapts in realtime to our mood. We instead work to build and curate playlists (though sometimes with AI assistance) for a huge variety of contexts, from needing motivation for a bleary-eyed 6am workout to setting the desired ambience for an intimate dinner party. There is a satisfaction to finding unexpected new music while putting in effort



to explore an enormously varied space of all the potential music in the world.

We are happy to revisit the same piece of art multiple times, if it has sufficient depth. And we are happy to put in effort to explore enormous spaces, when the act of exploring is satisfying and comes with the promise that sometimes we will find extraordinary beauty in that space. We put in effort to find and re-engage with art that we find emotionally resonating.

So what does focusing on this notion of emotional resonance mean to me when it comes to content generation in games? I'm not entirely sure yet. Maybe it means creating games that acknowledge and meaningfully engage with the machine's ability to create huge amounts of similar content (something that I think No Man's Sky is incredibly successful at) in a context that resonates with the player, instead of presenting machine-generated content as an infinite number of individual levels that become boring over time. Or maybe we could try writing new kinds of generators that aim to create smaller amounts of content that are individually more meaningful to players, maybe content or even games that have multiple layers or depth and complexity.

The joy that comes from hitting the generate button over and over to see something new is intense but fleeting, and then the joy turns to boredom, frustration, and disappointment. I want to stop thinking about content generators as being powerful because they can create a lot of things, and start thinking about ways to harness them for creating new kinds of emotionally resonant experiences.

*"Maybe it means creating games that acknowledge and meaningfully engage with the machine's ability to create huge amounts of similar content..."*



# Sure Footing

By Tommy Thompson, Supreme Science Overlord, Table Flip Games Ltd.  
@GET\_TUDA\_CHOPPA

*"While the core game worked, there was still a lot to be done to ensure the stability of the systems and the validity of the content it makes."*

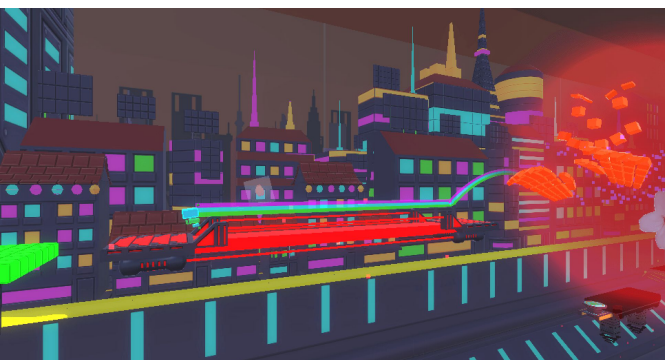
Hello to our fellow Procjammers - or is that Seeders? I'm really pleased to be able to report back on the current status of Sure Footing, which you may recall was the focus of a talk delivered by yours truly on behalf of Table Flip Games at PROCJAM in 2015. In our talk (which you can find on YouTube), I gave an overview of our infinite runner that transformed from a small research project into a fully-fledged game that we planned on launching. At the time we had just finished building our core procedural

there was still a lot to be done to ensure the stability of the systems and the validity of the content it makes. Ultimately we still needed to trust this thing to run for hours at a time without breaking, but also to create platforming levels that aren't going to prove impossible for players to traverse.

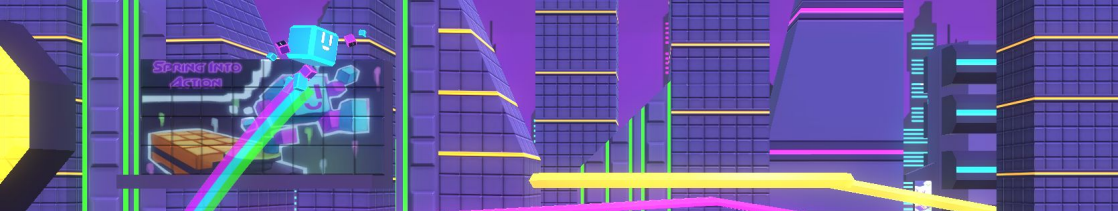
So here we are a year later and the game has come a long way. Firstly, we've completed the generative framework which now allows us to 'swap out' different PCG systems on the fly. Our game has two 'tiers' of PCG: one which considers the actions the players will be forced to make to survive and one which translates that action sequence into a playable 'sprint'. The real trick is that we can build multiple generators that look after each tier - and we now have almost a dozen generators either in development or currently in the game itself.

generation framework: a system derived from research in platforming games by my peers in the academic community. While the core game worked,

Given this is partially a research project, we've been testing and experimenting with the



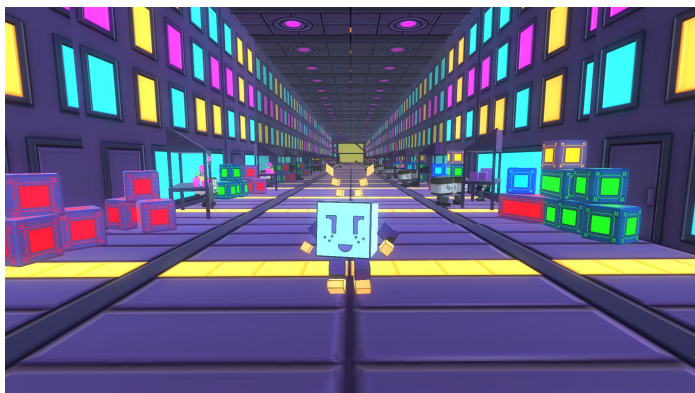




framework since we finished it and presented the game at the playable experience track of the 2015 conference in Artificial Intelligence for Interactive Digital Entertainment (AIIDE). We subsequently published our first full academic paper at the Procedural Content Generation workshop at the 1st DiGRA/FDG joint conference in August of 2016. We write about how our system works as well as quantify how expressive and flexible it is. This led to us adopting a large number of metrics in the game that allow us to effectively measure content as it is created and build an understanding of how long, how intense and how varied each level will be.

The research hasn't stopped there. We continued to devise new geometry generators reliant upon genetic algorithms to create new and unique interpretations of the action space. Also, we're using our level metrics combined with player testing to see if we can learn how to

'fit' the generative system around a player's performance. There are now over 20 parameters in the PCG system that allow us to define the starting difficulty of the game. We're keen to see if we can learn about our players to create new difficulty settings dynamically that will play against their weaknesses but

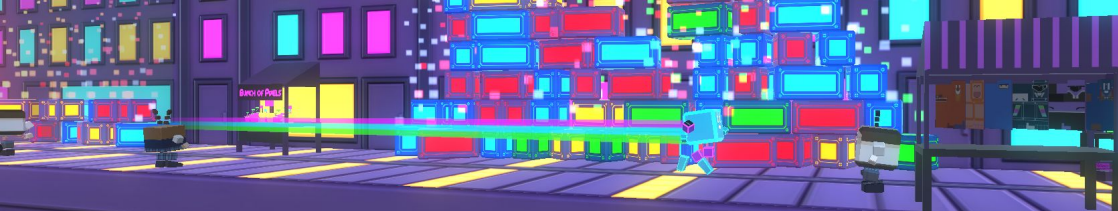


without being unfair. We have a long way to go before any of this is complete, but hopefully we can give you an update next year!

But enough about the research: what about the game? Well, why don't you see for yourself! Sure Footing launched in early-access on the itch.io Refinery in

*"We're keen to see if we can learn about our players to create new difficulty settings dynamically that will play against their weaknesses..."*





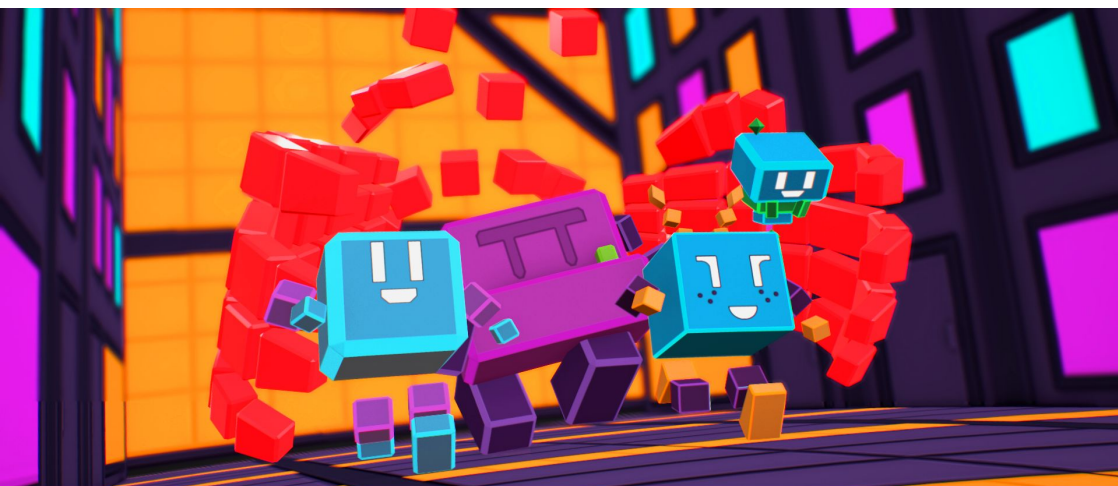
*"We knew that if we didn't launch the game soon then we never would..."*

September of 2016. Our plans had always been to launch the game either late in 2016 or early in 2017 but new research ideas as well as new gameplay ideas continue to emerge the longer we work on it. We knew that if we didn't launch the game soon then we never would - and given how popular it is when we take it to festivals in the UK we would be mad not to. As such, we've launched the game in early-access and are continuing to add new gameplay modes and features every month while talking with our players to take on their feedback.

It's been a crazy time for us since launch and while our community is small right now they have been overwhelmingly positive and supportive of our work. We're now looking not only to start migrating our work from the research build (aka the Branch of DOOM) into the public playable version, but also to run player-testing research with our community.

If you want to know more or get yourselves an early access copy, head over to:

**[tableflipgames.itch.io/sure-footing](https://tableflipgames.itch.io/sure-footing)**



# SURE FOOTING

# Sector A23

## Procedural world generation at run-time

By Jonas Delleske, Balint Mark, Joao Oliviera, Marek Skudelny, Lena Werthmann

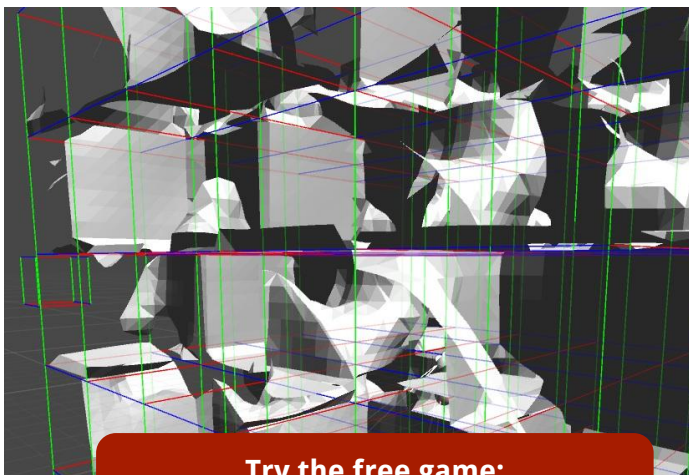
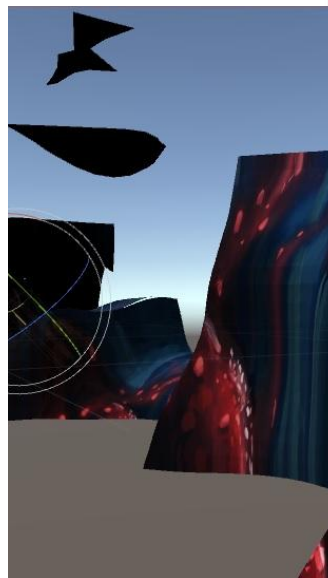
Imagine a world that only exists when you see it. Whenever you look around you see something completely new coming to life. This is the world of Sector A23. A beautifully generated cave system, filled with alien plants and mysterious creatures, giving room for endless exploration. Anything you see has never existed before and will stop existing as soon as you dare to look away.

This student project started with the simple idea that the world only exists when you see it. To build this game we started out creating an algorithm that would generate our cave system. Simultaneously we developed alien looking plants and strange creatures to live in this cave. Then we enhanced our algorithm to place those entities in the world. The world generation was also built in a way that we can add new world parts seamlessly to an existing world. So all we needed to do was delete the parts of the game world that the player does not see and generate a

completely new piece of world when the player is about to turn.

And thus the world only exists when you see it.

Summarized we successfully created a highly confusing game. The evil thing is that the player doesn't see when the world changes. And since everything looks alien and new the player doesn't even recognize that something is missing when he turns around. The change becomes part of the world and the world without orientation is accepted.



Try the free game:  
<https://wysiwyg.itch.io/sectora23>

# An Easy Way To Generate Fictional Alphabets

By @TearOfTheStar

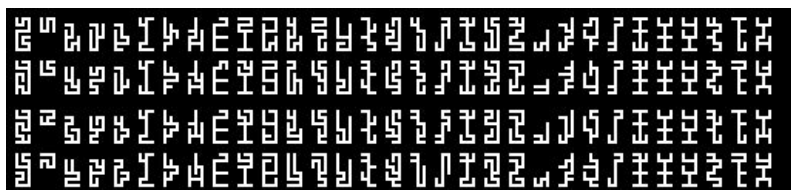
*"...that one can  
use simple  
binary masks to  
generate a grid  
based maze  
inside it."*

Maze generation is something every disciple in the field of procedural generation goes through at some point. The concept is extremely easy to understand (there are some excellent articles by Jamis Buck and Walter D. Pullen)\*, but it is an endlessly powerful tool in the world of procedural generation. While tinkering with maze generation, an idea came to me: that one can use simple binary masks to generate a grid based maze inside it.

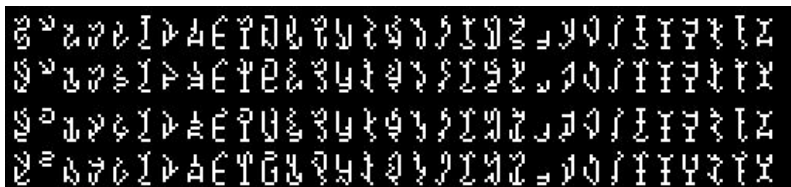
From this idea, the way to generate an alphabet was born. One makes/generates a binary mask (0\black - can't maze, 1\white - can maze here) like this:



... and generate small mazes using this mask. The result will look like this:



Because this maze generation is grid based, one can remove the corners, and this will result in a generated alphabet:



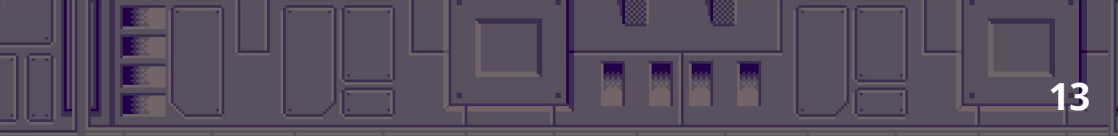




As shown here, only most limited of masks do not have variations.

So here's how to do it. My grid based mazes are generated with 1px cells, mostly because I like pixelart, so they look like that. However, one can generate them in any way one would like, even with bitmasking/curves etc.

\* <http://weblog.jamisbuck.org/>,  
<http://www.astrolog.org/labyrnth/maze.htm>



# In(finite) Content:

## Level Design for Games with Procedural Generation

By Oliver Carson

@OhCarson | [ocarson.itch.io](https://ocarson.itch.io) | [www.sizeablegames.com](https://www.sizeablegames.com)

The lure of procedural generation is an attractive one to game developers and game players alike. A game that can generate assets eases the amount of content developers have to create, and adds potentially limitless variance to aspects of a game.

Procedural content could be almost any part of your game, 2D, 3D, Audio, AI, Level Design, all to varying degrees. Let's examine, probably my favorite aspect of procedural generation, 3D world design, and how this links into level design. The first thing you'll need to figure out is what defines a level? Lets look at an example from one of my

current projects.

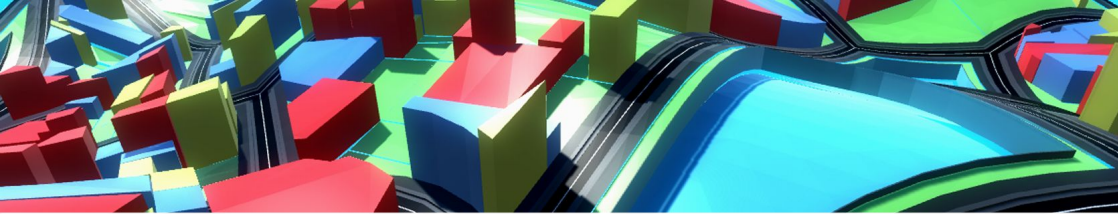
### Level Design in Dispatch! I'm in Pursuit!

Dispatch is a pet project of mine, it's what first got me into using procgen for 3D models. The gameplay in Dispatch sees hotshot rookie cop Lt Blaze cruise the city, stopping crimes and getting in hot pursuits in a future tech jet bike. Various buildings are littered around the city, criminal hideouts, raidable buildings, such as jewelry stores, and neutral buildings.

So the gameplay sees various heists happening in buildings throughout the city, and Lt Blaze has to race to each point and intercept getaway drivers before they escape to a hideout. To create this city procedurally, we need to **generate a road network** with traffic, a pavement for pedestrians, different building types, and navigation data for the AI. Phew that's a whole bunch of things!

*"Procedural content could be almost any part of your game, 2D, 3D, Audio, AI, Level Design, all to varying degrees."*





## The Tech Part

The road network is core to the gameplay of Dispatch, this can be simplified down to **a series of connecting lines**. The first thing we need to do is create these lines in 2D. One way to do this, is to distribute random points in a 2D space of a set size, with each point having a radius where no other points can be placed. These points are then connected using a **Voronoi diagram**. The voronoi diagram creates **2D polygons** out of these points. We can use these polygons to define the world geometry.

Each **edge** from these polygons, in gameplay terms, defines the road. We need to “grow” the road geometry from these edges. If we copy a polygon and shrink it we can create geometry for a road, however, we have to shrink this polygon down in a specific way, a standard scaling operation would look wrong, we need to **bring in each edge** from the island towards the center, and

discard overlapping line segments.

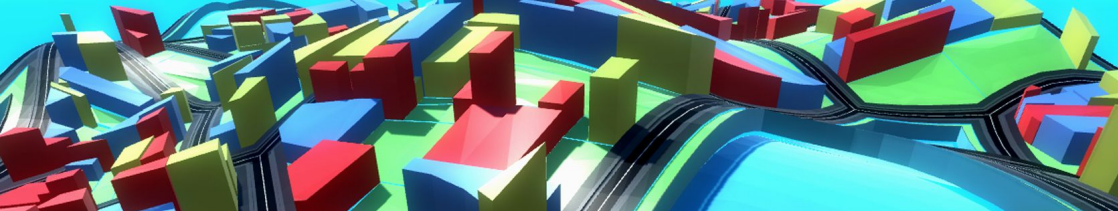
From there we’ve got enough data to begin extruding the road geometry with triangles.

We can create **additional polygons to create further details**, such as a pavement, a building or a field. Lets think about each polygon being an area. We could perform additional operations in the innermost “building” area to add more detail. Here we **subdivide** the building area and use these subdivisions to create separate buildings and back alleys. Each building is then assigned a type, such as jewelry store, or a criminal hideout.

As a benefit to creating all these lines, we can store them and use them for pathfinding. We also know the **context** of each area, so we can put cars on the road lines, pedestrians in the pavement area, back alleys might have more undesirables.

*“...it’s worth questioning if procedural generation is a good choice for certain assets.”*

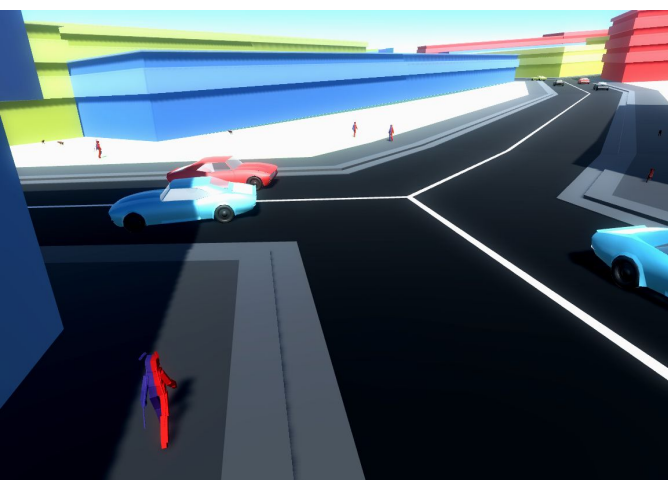




## When To Use Procedural Generation

The biggest issue with procgen is also its biggest strength, **variance**, creating a generator to make buildings in the city could be as simple as make a cube with some windows on it, but this could get pretty samey, creating large amounts of variance in any generator will undoubtedly take a longer time to code, it's worth questioning **if procedural generation is a good choice** for certain assets. Getting an artist to work on some discreet aspects may be the better choice for some of your content. It may take

significantly less time too. As someone who has spent a lot of time going down the rabbit hole of procgen, sometimes it's worth standing back from your work and asking, **would making a generator for this content be worthwhile to the player?** Can I afford the time? I feel procgen is generally best used for level design and random events, but **people are making some amazing stuff** in everything from, 2D, 3D, audio, and even narrative out there, so focus on what's important for your game and audience.



Procgen is a powerful thing, but is just one of your many tools to further your vision. Use it wisely and appropriately.

*Please look out for  
Cyglike a procedurally  
generated, cyborg  
animal hang gliding  
game in winter, and  
Dispatch! I'm in  
Pursuit next year.*





# Grow Your Worlds

By Luke O'Connor

We, as humans, are great at inventing new things to reach the unreachable. Boats, planes and rockets are all pretty good examples, but so are tanks of compressed air and crampons. Even something as simple rope is a pretty handy tool for getting someplace new.

Why then, when making games, would we start with the rope, the plane and the rocket, and then start creating worlds that need them? If all you have is a hammer, every problem looks like a nail. When it comes to procedurally generated levels, it can take a lot of effort to make sure the levels that pop out of the algorithm are traversable using the tools at hand. In other words, what we end up aiming for is a generator that outputs nails of different shapes and sizes. Which is kind of boring.

What if, instead of trying to ensure that your generator only pops out levels that are traversable given a set of movement mechanics, the

movement mechanics were designed around the generated worlds? Why not make our worlds first, then figure out a way to explore them. Better yet, make our worlds then allow the player to invent new ways to explore it. A more genuine experience of exploration awaits!



This opens up all sorts of possibilities, and allows us to generate more natural worlds. Worlds that just emerge from the universe inside our computers. Worlds that don't have the artificial limitations that gameplay may impose. Worlds that are as novel, and as unknown as our own, and that beg us to consider what is over the horizon, and how we might get there.

“...it can take a lot of effort to make sure the levels that pop out of the algorithm are traversable using the tools at hand.”



# Forska

By Todd Furmanski

Forska has been my current project for the last two Procedural Game Jams, and for reasons implied below, will be on my schedule for 2016 as well. The project can be described as a navigable, procedural landscape painting, where a user simply clicks on a static image to move, and the “painting” updating to that implied location. I mean to talk a little about why I developed the project, and go into a little bit of detail on how a few of its aspects work.

Various versions of Forska can be found at  
<https://tfurmanskigmailcom.itch.io/forska>

## A Sketchbook, Portfolio, Toolbox, and Zoo

Forska (Swedish for “Research”) is meant to be a place where I can implement different procedural techniques and see how they play together, without having to worry too much about a specific goal. Specific goals can be achieved later in external projects, where I can transplant code fragments and modify them to a more specialized end. Within Forska itself, each technique has a demonstrative effect on the virtual space, and I try to generalize it to make transplantation to a different project as easy as I can. Having them all in one place can be very handy as well. A lot of my experimentation with simulations and agent behaviors, for instance, proved slow going I realized I needed an interesting enough environment for the agents to inhabit, sense, and react to. Terrains, agent behaviors, graphical effects, and other dynamics can be hard to develop in a vacuum, so placing them all into one project seemed like the logical project for an ongoing coding jam.

I use Unity as the primary engine, which has helped me port code to a variety of different platforms, as well as allowing me to view parameters and world states easily. A certain amount of traffic between the Processing java libraries and my own venerable C++ codebase has also been known to happen.

*“Terrains, agent behaviors, graphical effects, and other dynamics can be hard to develop in a vacuum....”*



## Navigation

The idea behind Forska’s navigation is quite simple – you click on the image, and you’re teleported to where you clicked. In many ways this calls back to adventure games like *Myst*, but instead of having a limited database of images, I take the appropriate image using a virtual camera and a 3D space. It’s a simple matter of raycasting from the mouse cursor to the corresponding point in the landscape. I realize that many VR experiences have adopted a similar approach. This idea of “click to move” came from a need to quickly explore large virtual spaces, without slogging back and forth, or rocketing too fast past small destinations. With the paradigm Forska uses, taking a single step or walking a mile can both be done in one click. This approach has also proven to work very well with touchscreens and similar interfaces. I have watched far too many people struggle with a game controller while exploring a space at 24-60 frames per second. My countless hours of gaming in my youth have given me the dexterity to use a joystick or gamepad – many people have not had this tacit education. I wanted to experiment with removing this barrier of entry to exploring a virtual space.



## Non-Photoreal Rendering

The painted effect I give each rendered image starts with a typically 3D rendered camera shot, which then has a heavily modified blur shader applied to it. A separate, “noisy” texture input gives the blur a series of offset distances – the final result mimics brushstrokes, and it is this image input that controls stroke size, direction, etc. I tend to blur more in the midground, keeping the fore and backgrounds relatively detailed, since a faraway point of interest can be the same relative size on the screen as an object close to the camera, and faraway features can be obliterated if one simply does a “more distance = more





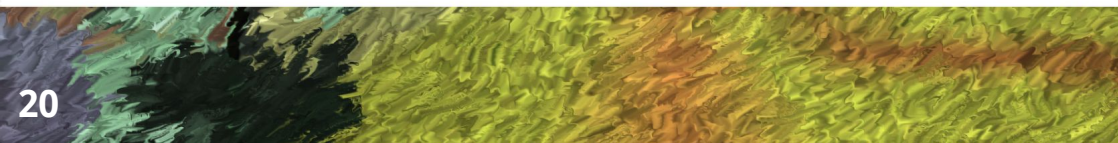


blur” calculation. Making blurs proportional to the sine of the distance can be helpful! In keeping with the “sketchbook” approach, I’ve developed several methods that mimic styles like oil paints, pastels, woodcuts, mosaics, and the like. Other procedural elements like terrain generation and dynamic skies give these shaders good subjects to work from, and can give even identical scenes their own sense of character.

*“...I mean to do more, each with a mixture of handcrafted and procedural components.”*

## Future

I mean to continue adding to this menagerie of techniques, as well as methods to explore and view them. I’ve done a few smaller works using components from Forska, and I mean to do more, each with a mixture of handcrafted and procedural components. I do not mean to add things like narrative, puzzles, encounters, or real time graphics to Forska itself, but I certainly plan to develop them for a variety of the project’s offspring!



# Simulations

By Sam Geen

@eegnsma

I am an astrophysicist, and I make universes, I tell people. Or... well, maybe. I suppose it's about the story I want to tell. Let me tell you a story.

Human history is often landmarked by the machines it creates. In earlier times, binary stars Mizar and Alco were used as a test of eyesight. Ancient astronomy relied on the optics in our own heads. Today, we put telescopes in space to observe galaxies billions of years old in light our eyes cannot see.

We don't limit ourselves to observing the skies. We remake them. Where the namesake of my first computer, Archimedes, once traced geometric proofs in the sand, today we build model universes in silicon. We trace gigayears of galaxy evolution in humming boxes, bits representing stars, interstellar gas and invisible matter leaping from machine to machine at the speed of light.

Here's an example. This week I used a small supercomputer

somewhere near Paris to simulate a bubble of hydrogen ions heated to ten thousand degrees by radiation from two massive stars. I wanted to understand a particular piece of how nebulae expand – what happens when the radiation increases tenfold as new stars are born. The question was whether an equation written in 1978 to describe this expansion still holds. It does.

“We don't limit ourselves to observing the skies. We remake them.”



An equation is a story. Each part is laid out in sequence, every variable clear in its role. In this example, I used a simulation to see whether this story was true or not – if we put all the same



*"A simulation is a landscape, unexplored until it is plotted, visualised, reduced."*



characters in the same setting, will they reach the same ending as the story?

A simulation isn't a story itself. A simulation is a landscape, unexplored until it is plotted, visualised, reduced. It is not a real landscape, but one we choose to generate, with its own choices and limitations. If our model for the violent death of massive stars is wrong, or we cannot resolve these vast explosions in our simulated galaxies, say, does our simulation tell us anything useful about them? As Deep Thought told Loonquawl and Phouchg, an answer without a question is pointless, and it's the question that takes the most thought.

Simulations have value because they are constructed to answer certain questions. They are expensive and time-consuming, both for the machine and for our limited time on the planet. We must, without biasing ourselves towards a certain result, ask what kind of story we want to tell before we ask the computer what the ending is.

As scientists we must ask

ourselves why we're doing what we do. Science is a toolkit for understanding the universe, and simulations are a powerful new tool in that kit, growing in power each time a new processor is printed by workers in Asia, each time designers push the physical limits of these intricate machines. It's tempting to see this as an end in itself, the march of technology accelerating us to a utopian future. But the only warmth a supercomputer gives is waste heat, dumping entropy into the slow heat death of the universe. Is this all we're doing, making bigger numbers until the universe grows cold and dark?

We must teach ourselves how to tell stories. Science is not an algorithm, a handle to be turned until the whole universe unfurls before us. It is a human act, people sharing narratives, trying to come to a deeper understanding. Scientific discoveries are human joys, whether it's finding a new creature deep under the ocean or trying to fit the cosmos into our heads.

Games, then. Computers can be sterile things. Procedural



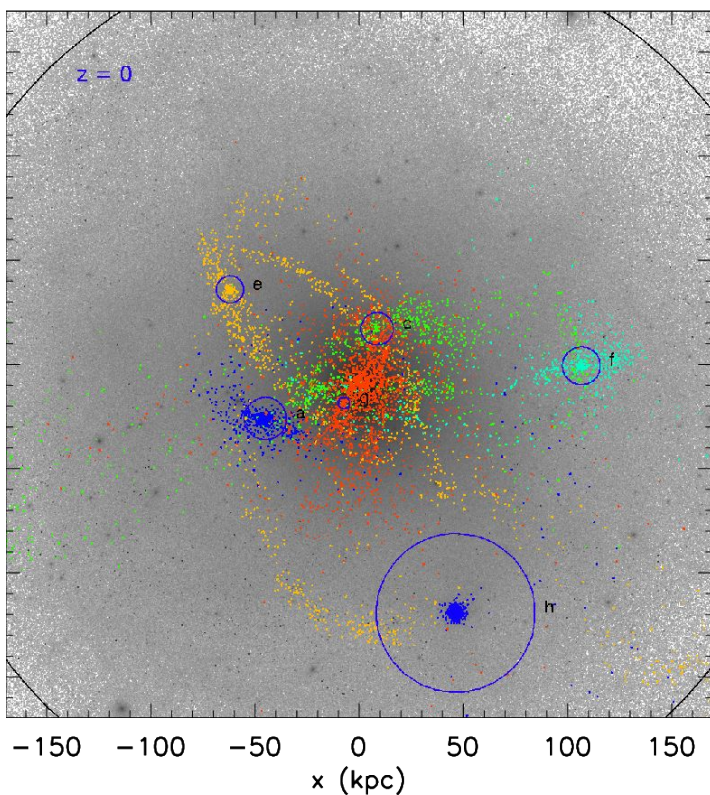
landscapes stretching for infinity with no life or purpose. But the computer was only ever a canvas, something for humans to impart meaning to. I follow a simulated person go about their day in a cityscape I built. I watch two siblings bond over their simulated person finding love in a dressing gown. I see a colonist nurse a raider back to simulated health, become friends.

Systems in games tell stories. The simplicity necessitated by simulations mean we must choose the underlying models. Do we want to tell a story of endless conflict, capital accumulation, ecological collapse, nationalism, fear of the other, shooting someone for the last tin of beans in the shattered world of our own making? Or do we want to deepen our understanding of the world around us, to foster beauty, to bring each other closer, to imagine worlds where exploitation, want and cruelty are not necessary and eternal. As Einstein said, “human beings are not condemned, because of their biological constitution, to

annihilate each other or to be at the mercy of a cruel, self-inflicted fate”.

The simulators have only interpreted the world, in various ways. The point, procjammers, is to change it.

“Or do we want to deepen our understanding of the world around us, to foster beauty, to bring each other closer...”



# Style Transfers

By Isaac Karth  
[procedural-generation.tumblr.com](http://procedural-generation.tumblr.com)

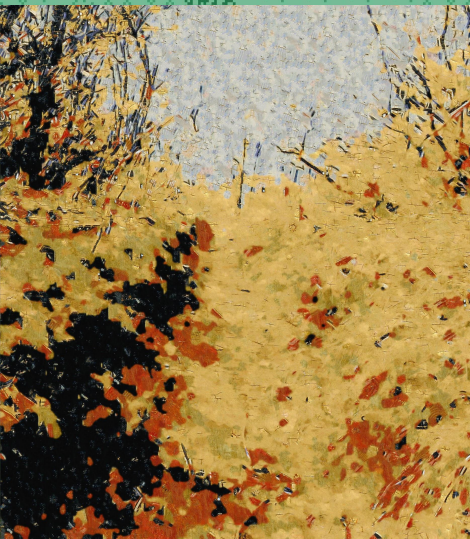
*Some of my recent experiments with the NeuralDoodle style transfer neural network.*

*"St Michael at the North Gate, Oxford, England, in the style of The Rocks by Vincent van Gogh"*



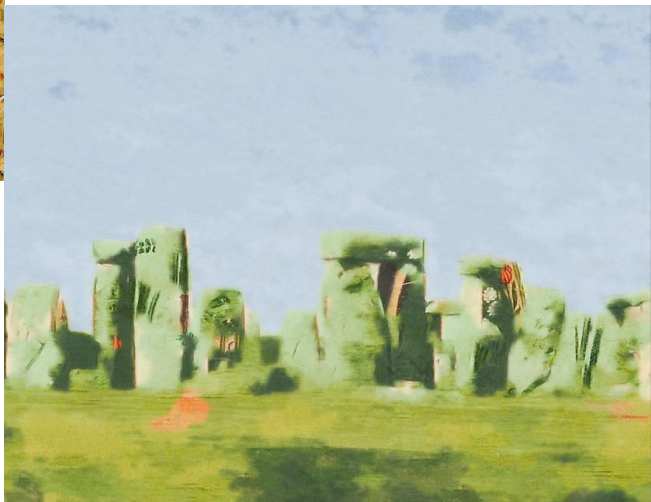
*"A comparison of the source photo and the result: a photo of the woods in the style of The Cemetery by Carl Fredrik Hill"*





*"A photo of the woods, in the style of Part of a Crucifix with the Ascent of Christ, 13th century, artist unknown"*

*"A photo of Stonehenge, in the style of a Mughal painting of Bibi Ferzana, c. 1675, artist unknown"*



*"A photo of a train station, in the style of View of Toledo by Aureliano de Beruete"*

# Space Noise Machine

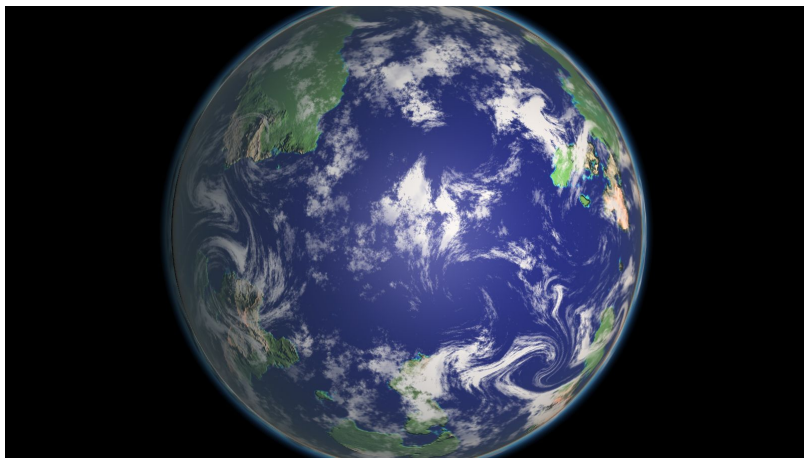
By David Murphy

*"The result is a  
splotchy noise,  
large patches of  
+1 with short  
edge fades to -1."*

I created my own noise library, named Space Noise Machine. It borrows heavily from the "module" concept of libnoise by Jason Bevins. I found libnoise was great, but I wanted to create my own. One of the reasons is my goal is a game made entirely of my own code. The second reason was that I thought libnoise could do with more modules.

The Earth-like planet you see was generated through a couple of spheres of noise. The ground, and the clouds.

The ground layer is produced by combining two Perlin Noise generators, one through a Ridged Multi-Fractal modifier and the other through a Fractional Brownian Motion modifier. This gives a nice rough surface with mountain ranges, we'll call it Noise A.



A third Perlin Noise is generated which is also put through a Fractional Brownian Motion modifier. This is used as a selector between two constant modules, -1 and +1. The result is a splotchy noise, large patches of +1 with short edge fades to -1. Or put another way, landmasses (with no detail) surrounded by beaches. Call it Noise B.



Noise A and B are then combined by using Noise B to decide whether to sample from a Constant -1 value (the water) or a value from Noise A.

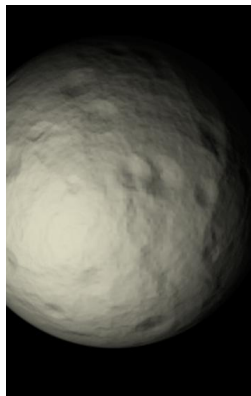
End result is randomly generated land masses with ridged mountains. The combination was done this way to avoid the constant "mountains in the middle" of the landmasses that you see when just taking Perlin Noise. It takes a lot more computational power, but the result is much better.

The cloud layer is generated in much the same way. A Perlin Noise is put through a Contrast Curve modifier and is used to generate clouds. A second Perlin Noise is used to act as a selector between a Constant -1 "no clouds" generator and the Clouds. The final, tricky part, is spiraling this noise around randomly distributed points by random amounts. This is what gives the clouds that swirly nature.

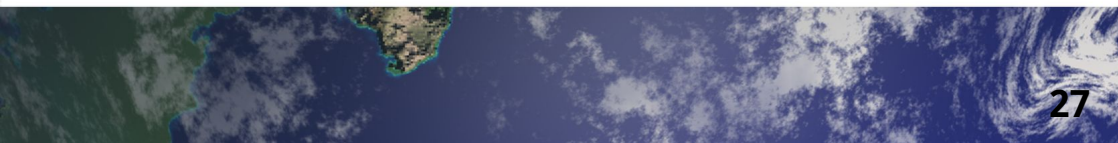
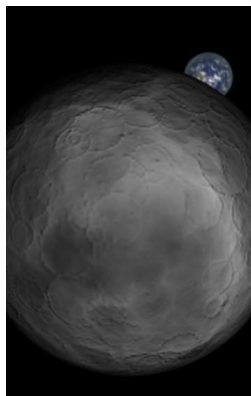
One of the most interesting things in this is that these aren't 2D images that are being generated, but rather cubemaps. This is done by all of the noise being generated in three dimensions and then being sampled along the surface of a sphere before being projected onto the cubemap. This results in no distortion at the poles.

The moons you see are low-res works in progress. The hardest part is the creation of craters. The size, distribution and nature of them is difficult; I've yet to create the modules to give them high ridges that quickly falloff into low valleys.

This is all a work in progress for a game I'm working on. As a programmer, not an artist, I needed to find a way to have pretty graphics without knowing how to draw. By letting my programming generate the art, it absolves me of a need for artistry and also gives me access to an endless amount of content.



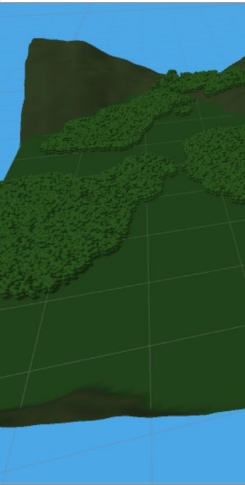
*"I needed to find a way to have pretty graphics without knowing how to draw."*





# Postmortem On Generating TOWNs

*By Gregoir Duchemin*



As my end-of-studies project, I teamed with four other students to release TOWN, our Tiny prOcedural World geNerator. You can check it out online (<http://delca.itch.io/town>); feel free to contact us at **pcg.town@gmail.com** if you have any questions.

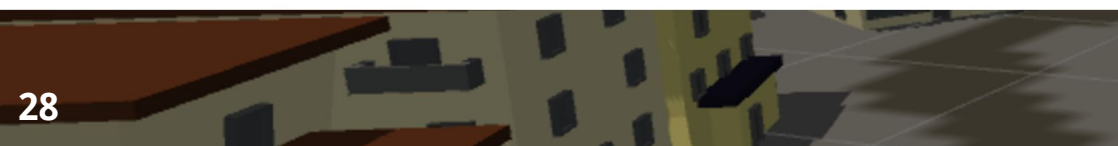
We were originally aiming for a village generator with 4 different themes to choose from, including "Countryside Village" and "Seaside City". Features like hills, lakes, forests and flat lands were required, and we used noise-generating functions parameterized to give us the terrain we wanted. Optionally, a river can pathfind its way from one side of the map to another.



To avoid floating buildings as much as possible, the village is placed on a large and flat area, upon which randomly-scattered points have been used to generate a Voronoi graph. Each face of the graph is assigned a lot type (deciding if it will be filled with buildings or decorative props) and some building templates, while roads are drawn along the edges. Other decorative features like a main road, street lights and utility poles were added to frame the village as part of a bigger world.

We also have on-the-fly generated music playing while visiting the village. Having not been involved in it, I sadly cannot say much about it, other than that it uses common phrases from jazz music and can produce really cool pieces with a bit of luck.

My role in this project was to fill the housing lots with buildings, a work I did in two parts. The first was writing a house model builder accessible via a small script language and a generator to create said scripts, and the second was about delimiting spaces in our lots to place the generated buildings. I learned a lot working on these tasks, and wanted to highlight some of my favourite takeaways from this project.





- **Do not be afraid to use simple methods.** The few articles I found on house and building generation were all using 3D maths to intersect solids. Since that was not trivial to implement from scratch, I decided to first try a grid-based approach. While the result feels blockier, it helped us when choosing a vision for the finished project (we tried to emulate a Godus concept art), and we eventually settled on it as an art style. Similarly, when we realized that our music sounded more "classical" (to untrained ears) when played with harpsichord samples, it removed the need for a more complex method.

- Like with handcrafting assets, **following a reference is a must.** As mentioned earlier, we used references and concept art from the Internet to define our visual style. This stays valid for non-visual things, like having an example script for a DSL or the outline of a complex algorithm to help staying focused on bite-sized pieces of the code base.

- **Setting up an out-of-code, easy way to test your generator** encourages more frequent tests when implementing new features; this was the reasoning behind my script-based house builder. That way, testing out new features and building styles is easier, does not require knowledge of the code base, and paves the way for presets and themed-based generation.

- **Breaking the parameters' limits** can lead to unexpected yet interesting outputs. By tweaking ours to force a very mountainous terrain, we managed to make a village spawn on top of a mountain, which was a scenario we wanted to avoid, and it turned out much nicer than we expected. We liked it so much it ended up in our presentation video !

- An organic look is nice, but can quickly devolve into a chaotic mess. **Adding some smaller, repeating patterns** helps with making the

*"By tweaking ours to force a very mountainous terrain, we managed to make a village spawn on top of a mountain..."*



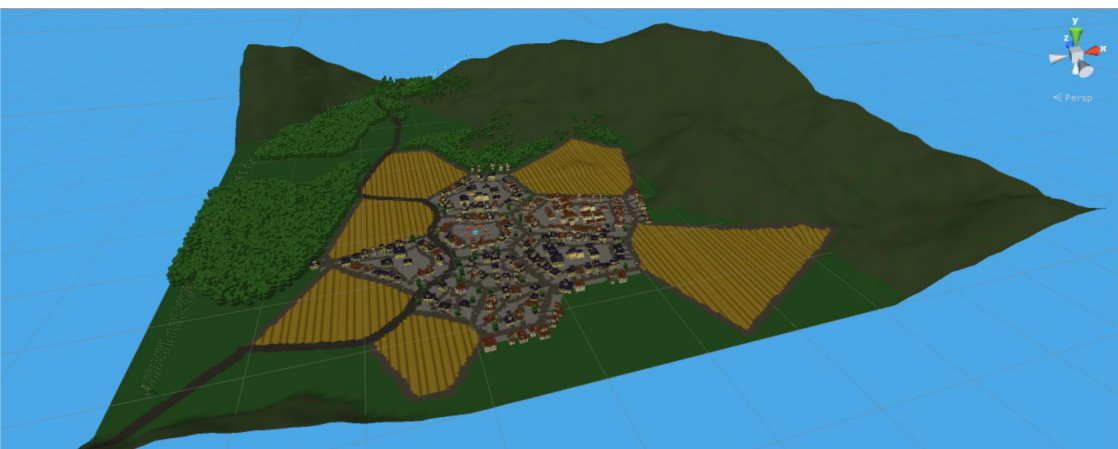


*"...a sheep field  
and handmade  
detailed houses  
to serve as  
known  
landmarks for  
the observer."*

output feel less totally random and is a first step toward constrained/themed generation. In TOWN, we contrasted the organic feel of a Voronoi graph with a grid-based placement for house in the middle of a lot. We also added some staple locations like a marketplace, a sheep field and handmade detailed houses to serve as known landmarks for the observer.

- As a follow up to the last point, **regularly getting exterior feedback is a must**, especially on a group project where your vision of the finished product is not the only one to shape the development. Had I worked alone, there would be much more grid-based placement in TOWN, which would have detracted from our aesthetic goals.

Overall, while there are things I would do differently were I to reimplement them, I think we managed to get the key points right and produced a result we were proud of. Maybe our major mistake was not having someone dedicated to make the visuals look nicer to the eye...



## Growing self representational life forms & some dusty software archaeology

*By Dave Griffiths*

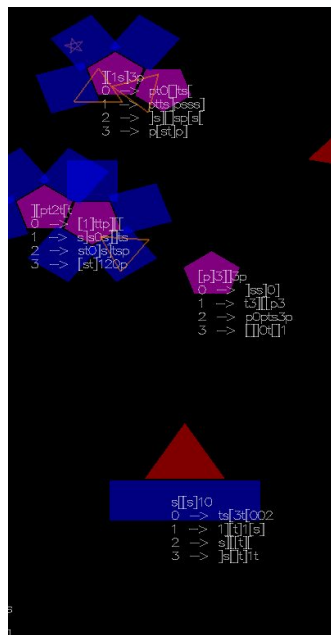
Sometimes you stumble over a dusty collection of source code you haven't thought about for years and can't even really remember writing. This article is about a bit of software archaeology, Moore's law and procedurally generating alien lifeforms.

GEO was a free/open source software game I wrote around 10 years ago. I made it a couple of years after I started working in my first job at a games company, and was obviously influenced by that experience. At the time I remember it was a little demanding for graphics hardware so I moved on to other things and forgot all about it, but it turns out the intervening years processing power has caught up. It was an attempt at a purely procedural game, with no assets at all – influenced by how demosceners built vast procedural worlds only with code. The main thing about GEO is that while being a slightly awkward 2D space shooter, the difficulty curve is


a side effect of artificial evolution that happens as you play, and learns from your actions.

The game is set in an expanding region of space inhabited by lifeforms built from component parts with different purposes – squares for generating energy, triangles for defence and pentagons which can be used to spawn copies when conditions are right. The lifeforms grow over time according to a genetic code which is copied to descendants with small errors, giving rise to evolution. The lifeforms have mass, and your role is to collect keys which orbit around gravitational wells in order to progress to the next level, which is repopulated by copies of the most successful individuals from the previous level.

Each game begins at level 1 with a population of randomly generated individuals, and the first couple of levels are quite simple to complete, as they mostly consist of dormant or







*“...a program  
trying to imitate  
something  
complex like a  
human brain, it  
really just  
represents  
itself...”*

Self destructive species – but after 4 or 5 generations the surviving lifeforms are the ones that have started to reproduce, and by level 10 one or two species will generally have emerged to become highly invasive conquerors of space. It becomes an against the clock matter to find all the keys before the gravitational effects are too much for your ship’s engines to escape, and the growth becomes too fast for your collection of weapons to ‘prune’ the emergent structures.

AI in games is mostly considered to be about emulating humans. What I like about this form of more humble AI (or Artificial Life) is that instead of a program trying to imitate something complex like a human brain, it really just represents itself – challenging you to exist in it’s utterly alien but consistent world.

I wonder why the dominant cultural concept of sentient AI is a supercomputer deliberately designed usually by a millionaire or huge company. It seems to me far more likely that some form of life will arise – perhaps even already exists – by accident in the wild variety of

online spambots and malware mainly talking to each other, and will be unnoticed – at first, and perhaps forever by us.

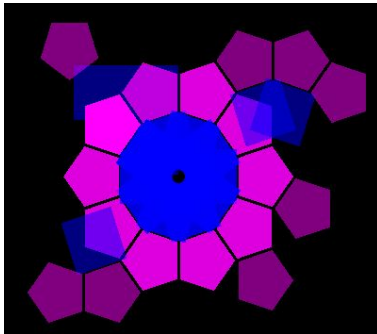
What I’ve enjoyed most about playing and tinkering with this rather daft game is exploring and attempting to shape the possibilities of the Artificial Life while observing and categorising the common solutions that emerge during separate games – cases of parallel evolution. There is a fitness function that grades individuals which is used to bootstrap the population before they are good enough to survive (e.g. they get points for simply shooting at the player or generating an energy surplus), but most of the evolution after the first couple of levels tends to occur ‘naturally’ while you are playing. The species which takes over is the one that manages to reproduces most effectively, defends itself and repairs damage the best.

### **‘How it works’**

Each individual in the population carries around a text description of itself, a Lindenmayer system, which contains an axiom (it’s starting condition) and 4 replacement



rules. We start with the axiom and the rules are repeatedly run on the output string to 'grow' the life form. This is an example of a successful organism "grown in the wild" and it's L system description:



```
axiom: "[[sp3ss"
'0' → "[p3t]]]"
'1' → "t][][]]"
'2' → "2[s]t[tt"
'3' → "ps[spp0s0"
```

At each growth step, the numbers are replaced by the corresponding strings – which can contain their own numbers and provide recursive self similarity. These are the first 5 growth steps for this organism, starting with the axiom:

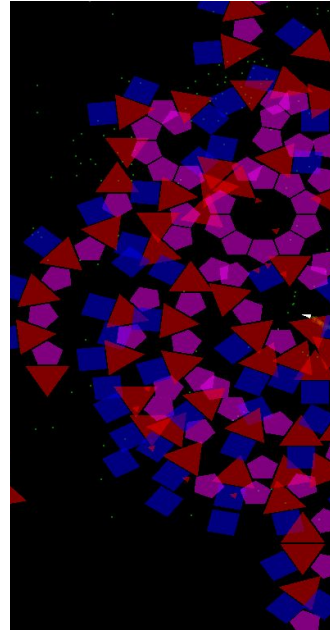
```
0: [[sp3ss
1: [[spps[spp0s0ss
2:
[[spps[spp][p3t]]]]s[p3t]]]]ss
3:
[[spps[spp][pps[spp0s0t]]]]s[p
ps[spp0s0t]]]]ss
4:
[[spps[spp][pps[spp][p3t]]]]s[
p3t]]]]t]]]]s[pps[spp][p3t]]]]s[
p3t]]]]t]]]]ss
```

These strings are then parsed to convert them into structures: 't','s' and 'p' represent triangle, square and pentagon. When one of these are found the parser searches for following blocks of characters enclosed by square brackets. These are attached to the sides of the shape in order to provide a tree topology.

There are many things that could be done to improve or expand this game, make it 3D, get it on different platforms and so on. I've recently uploaded the source here with some tweaks to make it easier to compile:

<https://gitlab.com/nebogeo/geo>

Let me know if you get anything interesting out of it, or develop it in new directions.



## About the author:

Dave Griffiths is an award winning game designer, programmer and livecoding algoraver based in Cornwall. In 2014 he co-founded Foam Kernow, an independent research institution for exploring uncharted regions of art/science and designing speculative cultures. Previously he worked in the games (Sony Europe) and film computer graphics industry (Moving Picture Company), and has credits on feature films including Troy and Kingdom of Heaven.

# Chance A: Me

By *Ciro Duran*

[www.ciroduran.com](http://www.ciroduran.com)

*"It seems that ideas come from what you most usually do or live."*

I'm a frequent commuter since I was old enough to take public transport by myself. Subway, bus, train, you name it. Some people hate the idea of taking public transport, but I sincerely love it. Most of my game ideas revolve around the idea of experimenting with traffic, and that hit me once a friend pointed that out to me. It seems that ideas come from what you most usually do or live. This is a realisation that I can live with, and it still keeps giving me ideas, even if I can't get them together for a game.

Ad written: Smiling beauty traveling on the Northern line on Friday morning. Coffee sometime? - Cyan jeans woman  
Ad written: Long-haired beauty with a paper on the Victoria line at ((hour))pm on Saturday. Meet up at the pavement anytime? - Red hoodie woman  
Ad written: Something about your nose got me. Fancy having smoothie one morning with me? - Green dress woman  
Ad written: Strong beauty with a tablet on the Bakerloo line at ((hour))pm on Saturday. Whisky sometime? - Yellow suit woman  
Ad written: Tall beauty who wears the green hoodie and catches the 21:22 train from Cannon Street. I would love to see you again. - Stunning woman in yellow onesie  
Ad written: Something about your nose got me. I would love to see you again. - Yellow dress woman  
Ad written: Stunning beauty traveling on the Northern line on Saturday dawn. I would love to see you again. - Red hoodie woman  
Ad written: Bold beauty who wears the red pullover and catches the 14:19 train from Hampshire. Do you fancy a beer sometime? - Wizardly woman in black jacket  
Ad written: Skinny beauty with a paper on the Central line at ((hour))pm on Monday. I wish we had had an extra second - Green suit woman  
Ad written: Something about your neck got me. Coffee sometime? - Yellow hoodie woman  
Ad written: Something about your eyebrows got me. Any chance of a date? - Red dress woman  
Ad written: Hopeful beauty who wears the green dress and catches the 14:15 tram from West Croydon. I would like to see your lovely eyebrows again - Green pullover woman  
Ad written: Something about your shoulders got me. I wish we had had an extra second - Handsome woman in blue hoodie  
Ad written: Round beauty with a book on the District line at ((hour))pm on Sunday. I wish we had had an extra second - Short-haired woman in cyan jacket  
Ad written: You are a smiling beauty in their late fifties who wears a blue pullover. I would love to see you again. - Red onesie woman

One of the most curious things I've found during my commute is the "love connection" section of the free newspapers I get before I board the train. This section contains very short messages of people that saw other people and wish to see them again to start talking. I've seen teenagers reading the section messages with funny voices in order to pass the time in the train, so I guess getting your message out there like that seems kind of desperate. Still, some of these texts are clever, some sweet, mostly are not specific enough to be creepy; you could even argue they are the product of some intern's mind and their desires, written in a corner at the newspaper headquarters.

Anyhow, I started wondering on a game that happens while commuting but it's not about traffic, but rather about the people that commute regularly. There are some unwritten rules about commuting: do not chit chat too long with someone who is visibly annoyed by your attempts to talk with, do not stare too long at people, DEFINITELY do NOT talk to someone who is wearing their earphones, among others.



At the same time, if you're looking to connect with someone, you need to somehow go around these rules. This forms the basis for a short game, for which we can add a bit of procedural generation to generate stories. What if the love of your life is with you in the same carriage at this very moment? They're sending a message to you, you just have to figure out who is telling you that, and try to find their gaze. Look into them for very little time, and they won't find out, look into them too much, and you'll scare them away.



I'm currently experimenting to see where this premise leads to. In the technical part, I'm using OpenFL for drawing some faces and animating them, and I'm using Tracery (<http://tracery.io>) for building the messages. I'm specifically using a Haxe port I made (<https://github.com/chiguire/traceryhx>) from the original Javascript.

The game generates a character from a series of parameters (you can see an example here - <http://tinyurl.com/seedschance>), and then it would build a grammar from those features to build a message. Since the person must describe you, you should also create your own character. You can notice that the hair is still work in progress. :)

The idea of the game is to have very simple controls, just move the mouse and click a button, using the gaze as the main verb in the game, but this is all still experiments. For now, I just have a simple way to display faces, and a way to generate silly descriptions. In order to get the ball rolling, I fed the Tracery grammar to a bot with these descriptions, which you can see at @chancea\_me thanks to Cheap Bots, Done Quick! (<http://cheapbotsdonequick.com>). Hopefully the bot will explore some ways a relationship could start (or crash and burn).




I hope you have fun, and you can find some inspiration on your bots/procedural generation/stories in your day to day.



# Overworld Forever

*By David Morrison*

Overworld Forever is a tile-based adventure game created by training a level generation system on the overworld map from The Legend of Zelda. The game uses an n-gram based approach that can create overworlds that are statistically similar in layout to the original game. By increasing and decreasing the length of the n-gram, the system can be made to produce maps that vary between having no coherence between adjacent tiles to ones are so constrained that they are identical to the original Zelda map.



Somewhere in the middle of these two extremes are overworlds that are similar enough to the source map to be playable while providing enough variation to be interesting. They are often hard or impossible to traverse with paths that lead to nowhere, rivers that flow into deserts then stop and horizontal bands of hedgerows and boulders where the system gets stuck in probabilistic cul-de-sacs.

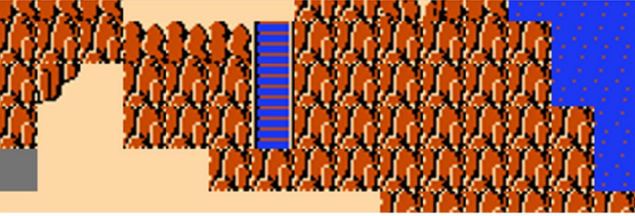
Like all Machine Learning based generative algorithms, the system works in two phases; first, it trains, then it generates. In the training phase, the system works out the probability that

each tile will appear at the end of a sequence of other tiles in the original map. Any sequence of tiles can be represented as a string of their indices concatenated together. These can then be put into a hash table. Each sequence hash is keyed to a list of possible successor tiles. The algorithm iterates over the map and every time it encounters a sequence, it records what the next tile is and places into that sequence's successor list.

To generate overworlds, the system puts down a random tile at the map's origin, then looks for the sequence containing only that tile in the hash table. It then selects the next tile from the list of successor tiles for that sequence. This process continues until the system has generated enough tiles to fill the new map.

In the future, I'm hoping to expand the game to include dungeon levels and limit the space of possible overworlds to ones that satisfy basic playability constraints like making sure the player can walk to every room on the map. Perhaps hybrid systems that blend grammar-based approaches with machine learning might be a good way to





train dungeon generators with similar flows to the original Zelda while making sure they can be traversed and completed.

The n-gram technique described above can be applied to any tile based game or image. Statistical learning approaches such as this provide designers with new ways to explore the parameter spaces around their designs without explicitly formulating them as generative systems. There is the potential to integrate them into level editors, allowing designers to train their design tools on collections of their previous work. The broader family of techniques is not limited to grids. For example, stochastic graph and shape grammars can be trained on level topologies. This increases the kinds artifacts and forms that can be generated to include most game genres.

Machine Learning also opens up the possibility discovering structures and patterns inside existing games and directly transferring them to future ones. The topology of the levels in a game like Pacman could be used to train a model for generating first person shooter levels, for example.

Perhaps that's getting a bit far out but even relatively straightforward techniques can yield unique and interesting results based on existing games and content that's just sitting around waiting to be mined!

Overworld Forever is implemented in Processing using sprites and map data stolen from The Legend of Zelda. It is available on Github at <https://github.com/davemor/overworld-forever>.

*“Machine Learning also opens up the possibility discovering structures and patterns inside existing games and directly transferring them to future ones.”*



*David Morrison is a research assistant in the St Andrews Human-Computer Interaction Research Group. A long time ago he used to work in the games industry.*








# Moai

By *Eggy Interactive*  
[www.moai-game.com](http://www.moai-game.com)

## An Introduction



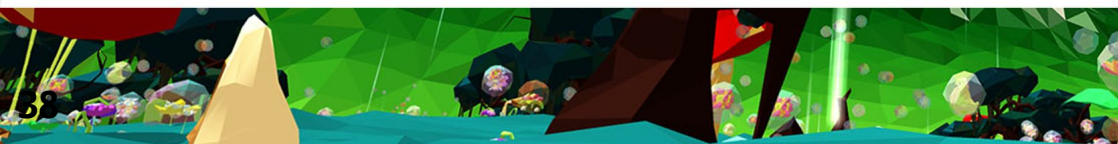
Moai is a procedurally-generated low-poly exploration game where you play as a moai, or a sentient stone being, with the power of infinite patience, allowing you to fast-forward time and watch trees sprout and days and nights pass. This game was made as an undergraduate senior project at the University of California, Santa Cruz by a team named Eggy Interactive.

Our vision from the beginning was to create a beautiful, vibrant, and vast world that players could lose themselves in. To achieve creating a world with the size and complexity that we wanted, we turned to something called procedural generation. This basically means that before the player hits PLAY, the world doesn't exist yet. The game will create the world on the spot as soon as you hit that button, creating a world based on pseudo-random numbers and values that the computer generates, resulting in something completely different every time the player starts a new game.

## Creating a World

The world created by the game comes with many system such as a day/night cycle, weather, vegetation system, and more. The most fundamental system is the terrain generator. The world is generated in square units we call chunks. The topography of these chunks are determined with perlin noise maps, which decide the height, kind of topography, and biome of each chunk, resulting in various formations like hills, valleys, and mountains.

What about the objects? If we places objects randomly into the world, the system would sometimes end up putting all the trees in one spot, similar to how sometimes when you flip a coin multiple times, you somehow get heads every time. Even if the objects were placed spaced out randomly, the area would look more like a messy room than a forest. Instead, we divided up the chunk into smaller





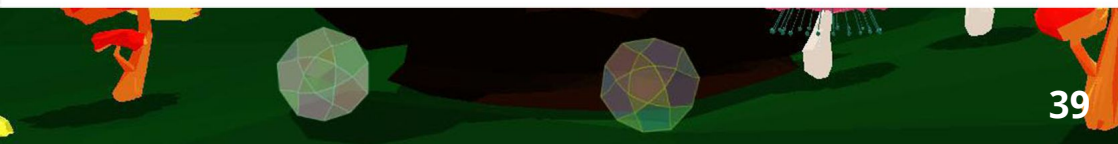
areas, then designate a spot to place a cluster of objects in those areas, and then place objects in a random space inside a circle around that spot. This way we get nice clusters of objects and makes the area look more like a natural forest.

### A Natural Low-Poly Look

When it comes down to it, there was just so much we could leave to the system to generate. That being said, not every single aspect of the world was left for our generators to decide. While the “when’s,” “where’s,” and “how’s” were decided by the generators, the “what’s” were designed by us. One of the key elements the system needs to know when deciding what to put in each chunk was what biome that particular chunk is going to be. We designed eight unique biomes that the system can choose from, each with their own set of weather, vegetation, and scenery. These things all need to be manually designed by our art director, who decided on the low-poly aesthetic.

Low-poly is a very popular aesthetic taken on by many games - especially indie games because of its simple yet elegant look that is easy on the eyes. During prototyping, we attempted the “super simple” style that many low-poly games opt for, but while the visual were very nice and clean, we felt the hard edges made the world feel less vivid, natural, and “alive,” so we tried a different approach. Instead of using the polygonal shapes to define just the generic shape of each object, we also used the vertices and polygons to create texture in each object, adding more detail to object and giving it a more natural look while still keeping the elegant style of the low-poly aesthetic. We have to give props to our very talented art director who somehow found the perfect combination of order and chaos, incorporating a sense of flow for the eyes to follow in each design and animation. Speaking of animation, the world actually grows right before your eyes. Every large plant is animated to grow as you fast-forward time, allowing you to witness the life cycle of entire forests. Additionally, the vegetation and scenery are

*“Instead of using the polygonal shapes to define just the generic shape of each object, we also used the vertices and polygons to create texture in each object...”*



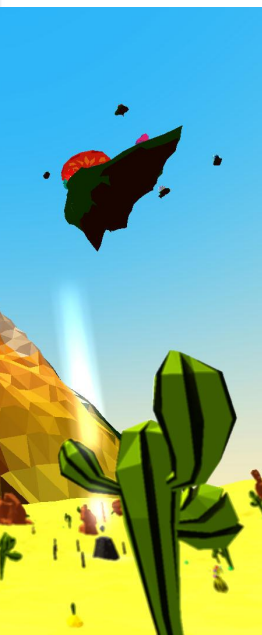


programmed to each have their own ambient animations, such as vines swinging, flowers waving, and water stirring. When you stand still, the world isn't frozen, but full of life.

With all of this, we've made a vivid and beautiful world with absolutely nothing to do in it. In fact, with procedural generation, it's very easy to make something extremely pretty, but also extremely boring. Our biggest design obstacle was now: what do we do with this world? We've made the interesting to see; now how do we make the interesting to do?


### **The Interesting to Do**

Our solution to that is Points of Interest. The primary goal of points of interest is to promote exploration - or in other words, get the player to walk around the pretty world we worked so hard to make. In addition to placing the vast amounts of vegetation and scenery, we also have our system strategically place shrines and obelisks, which have cryptic symbols on them used for puzzles. These structures are made to be very noticeable - they have glowing beams and react to player interaction. These are meant to lead the player from point to point in the world. Our biggest points of interest are giant floating islands in the sky. We've designed these islands to be the ultimate end-goal of the game, so they're really high in the sky and you can see them from practically anywhere, acting as a visual goal that the player strives to reach.



All of these things that we've designed and generated amounts to this vast and beautiful world that the player can explore endlessly. Moai was a game that was designed over a short period of five months with a small team of five people as an ambitious senior game design studio project, which is part of the University of California, Santa Cruz undergraduate Computer Science, Computer Game Design program. The current release of the game is only the





beginning of the vast vision we have for this game. During the two quarters we had to make this game, we had to scrap many amazing ideas because we had many deadlines to meet in an unfavorable amount of time, but now that the idea has become a tangible, working game, we're excited to continue working on it and turn it into the vision we've always wanted Moai to be.

Until then, the game can be downloaded at [eggyinteractive.itch.io/moai](http://eggyinteractive.itch.io/moai). We hope that you have as much fun playing it as we did making it!

### Moai by Eggy Interactive

Brian Lin – Creator, Designer, Programmer

Yunyi Ding – Art Director

Ryan Lima – Developer, Programmer

Nathan Irwin – Producer, Programmer

Anderson Tu – Composer, Audio Designer, QA Coordinator

[www.moai-game.com](http://www.moai-game.com)

[eggyinteractive@gmail.com](mailto:eggyinteractive@gmail.com)

*“...we had to scrap many amazing ideas because we had many deadlines to meet in an unfavorable amount of time...”*



MOOI



# PCG without a Computer: Combinatorial Literature

By Davide Aversa  
@thek3nger

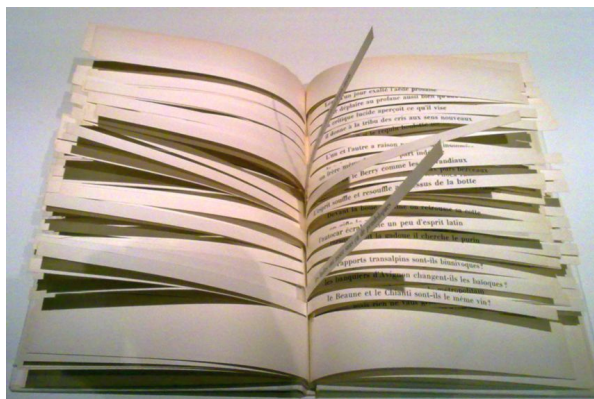
*"The reader can, therefore, 'generate' a different poem by changing each of the 14 verses with one of the 10 variations."*

For us computer scientists and game developers, *Procedural Content Generation* is directly connected with computers and algorithms. It seems such a modern thing!

In reality, the exploration of the "combinatorial nature of art and human thoughts" is much older concept. Probably, the most interesting and old writing on "PCG" is the doctoral dissertation of Gottfried Leibniz, *De Arte Combinatoria* (On the Combinatorial Art) (1666) in which he exposed the main idea that "all truth are nothing but combinations of a relatively small number of simple concepts".

Even if this small idea was always in the back of the head of the most daring artists, we have to wait until 1961 to see the first literary work that we can define a PCG opera. That year, *Raymond Queneau*, a French novelist and poet, published the book *Cent mille milliards de poèmes* (Hundred Thousand Billion Poems). The book is composed by just ten sonnets of 14 verses, but it was printed such that each verse of the sonnet is on a different paper strip. The reader can, therefore, "generate" a different poem by changing each of the 14 verses with one of the 10 variations. At the end, the book contains  $10^{14}$  different combinations, hundreds thousand billions poems, precisely.

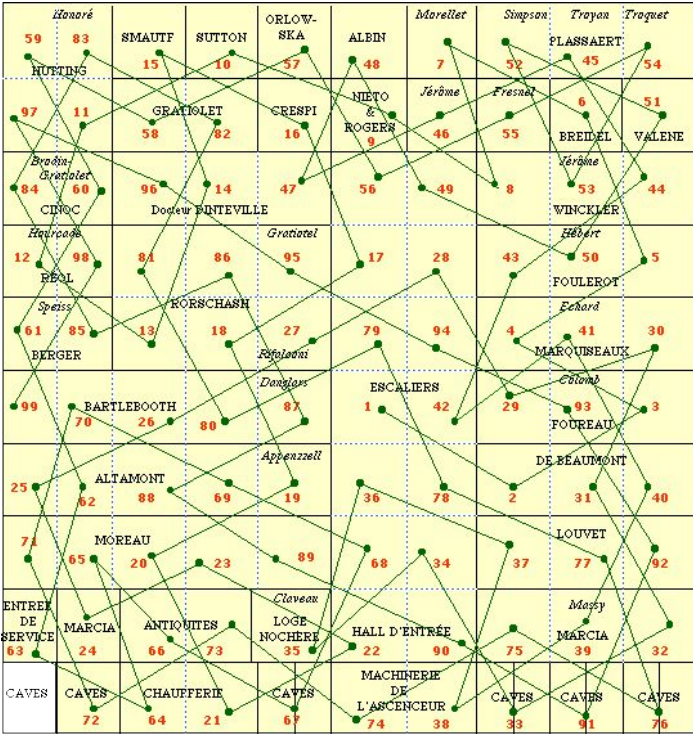
Probably, Marc Saporta, another French writer, thought that  $10^{14}$  was not enough because in 1962 he published the book "*Composition n° 1*", a book composed by 150 not-numbered pages that can be shuffled at will by the reader producing  $150!$  (factorial) different books.



This was just the beginning of a literary movement called **combinatorial literature**, in which authors used math and combinatorial generation as a tool for inspiration. The most emblematic author probably was Georges Perec who creates a complex system referred by himself as “a machine to inspire stories” for the book *La Vie mode d'emploi* (Life, a User's Manual). The book tells the story of a big building with 10 floors and 10 rooms per floor (imagine a 10x10 square). The narration starts from a room and continues with L-shaped movements (just like the Knight in Chess) from room to room until covering all the rooms *but 1* (the basement).

Moreover, the book contains 42 lists of objects such as, emotions, animals, countries and more arranged according several **Graeco-Latin squares** (a 10x10 arrangement of pairs of different lists such that every row and every column contains each element of one list exactly once, and that no two cells contain the same

ordered pair). These squares are then explored “randomly” by the L-shaped narration producing a list of objects that the author have to include in the current chapter. I know, it is quite confusing right now. But I really encourage you to look more in details on the mathematical structure of this book! It is worth your time.



*"...each part of the description can be exchanged with each other so that "the reader can create its own path in the book."*

Another author fascinated by the use of mathematical rules to generate novels was Italo Calvino, an Italian novelist. The influence of the combinatorial authors is clear in book such as *Le città invisibili* (Invisible Cities) in which the author (as Marco Polo) describe 45 cities according 9 thematic groups and in such a way that each part of the description can be exchanged with each other so that "the reader can create its own path in the book". Or in *Il castello dei destini incrociati* (The Castle of Crossed Destinies) in which the author use a deck of 73 tarots arranged in such a way that reading each line (from right to left or left to right, but also from top to bottom or from bottom to top) describe one of the 12 stories narrated in the book.

You probably have noted that all the author I mentioned are French (except for Calvino), but they have another point in common. They all (except for Saporta) belong to the same literary group: the Oulipo (Ouvroir de littérature

potentielle, workshop of potential literature). If you are interested in this complex experimentation with the narrative and the human language you definitely have to take a look to the Oulipo's authors. Ah, by the way, the group is still on activity and has a nice website (<http://oulipo.net/>). Check this out.

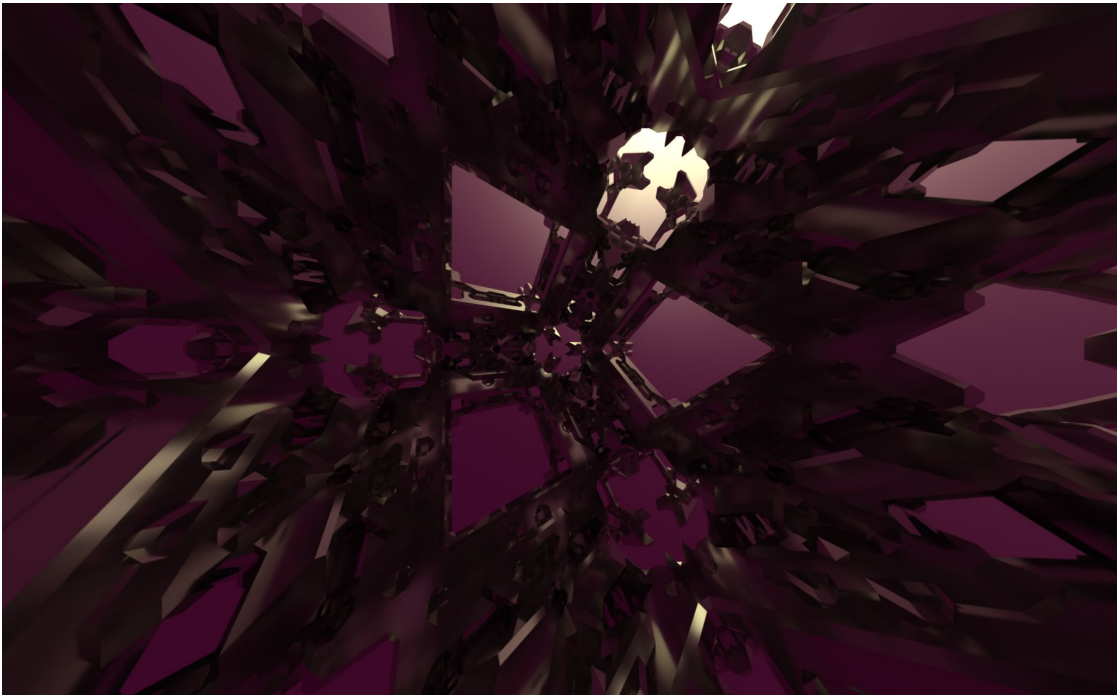




# 3D Cellular Automata

*By Kevin Chapelier*

I started investigating offline rendering of 3D cellular automata after my work on öde ( <https://kchapelier.itch.io/ode> ) for PROCJAM 2015 which used 3D cellular automata to create abstract skyscraper-like structures in a huge simplex/perlin noise desert.

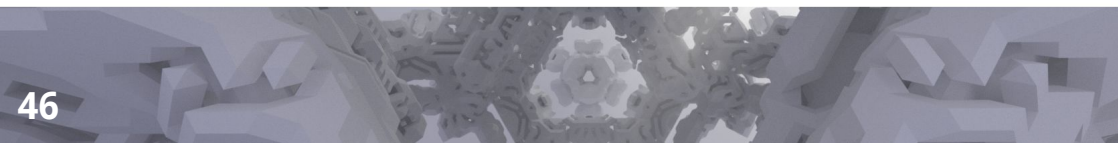
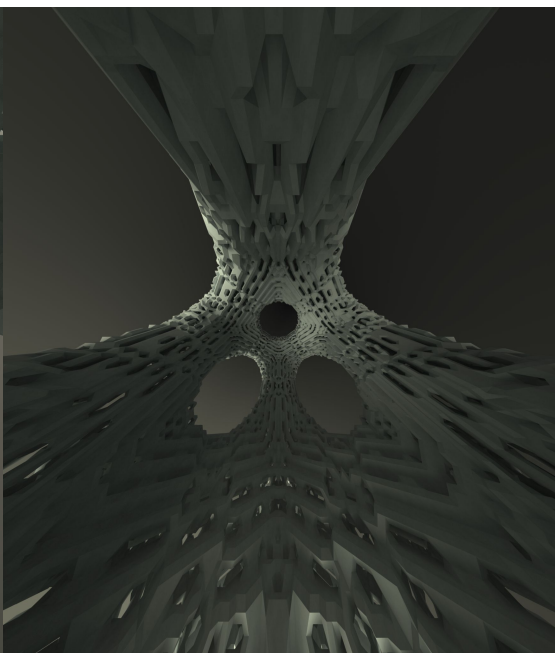


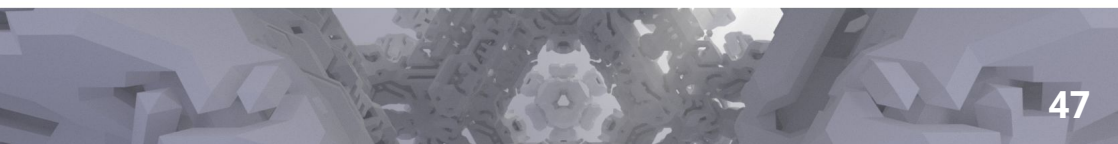
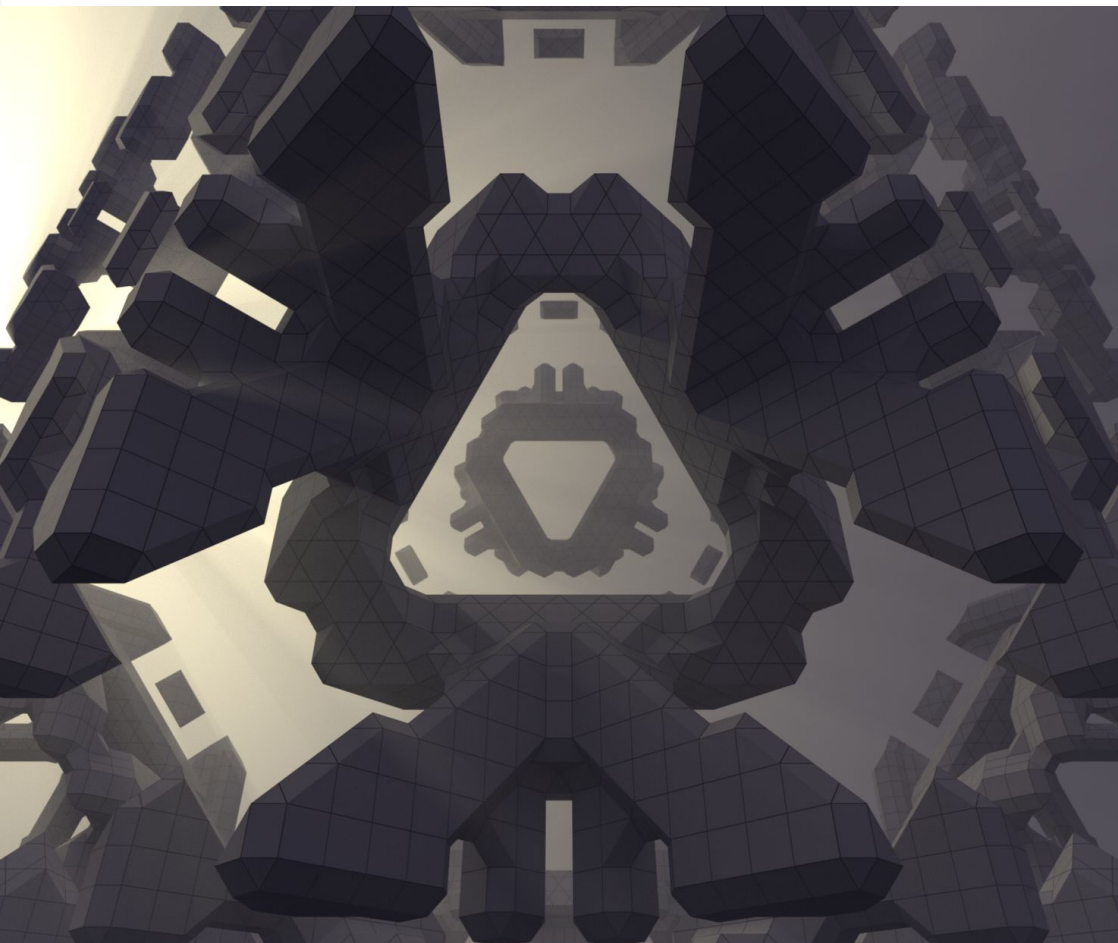
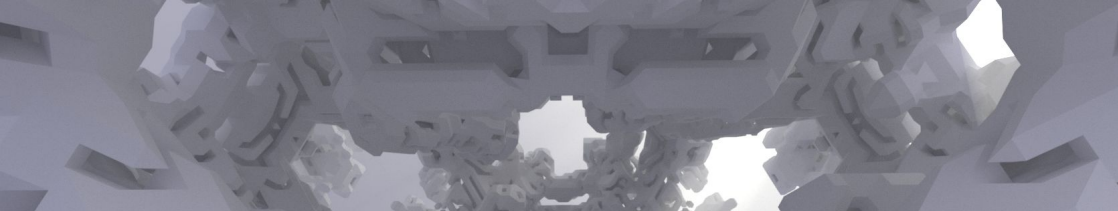
Nowadays I use MagicaVoxel and a custom command line interface tool ( <https://github.com/kchapelier/cellular-automata-voxel-shader> ) to modelize the volume on the GPU by applying several CA rules iteratively. In this particular workflow, each cellular automata rule can be thought of as a simple volumetric hue or paint and, with enough practice, the user develops a general intuition of how those paints will behave when mixed together.





Working directly with MagicaVoxel (developed by <https://twitter.com/ephtracy> ) offers a lot of advantages: quick previews of the volume while working on it, the ability to 'undo/redo', a path tracing rendering engine with a lot of options including a marching cube rendering which is perfect for cellular automata and the tool is frequently updated with new features.





# ProcEngine: An Open Source Procedural Map Generation Engine

By Ahmed Khalifa

Every time I start thinking about designing a roguelike or a game that use procedural generation for maps, I start googling to see what are the different techniques generation techniques. After selecting the best one, I start writing a code for it from scratch or copying it. This process is tiring and cumbersome especially during prototyping phase. In prototyping, I need just to test the idea as quickly as possible. The main problem when one of these ideas depend on procedural generated maps. After creating couple of roguelike prototypes, I couldn't take it anymore. I decided to write my own library that I can use it in prototyping. I called this library *ProcEngine*.

ProcEngine is an open source procedural map generation engine that allow the user to select from bunch of different generating algorithms and tune them. ProcEngine is inspired by Nicky Case (*Simulating the world (in Emoji)*) and Kate Compton (*tracery.js*). The current version of ProcEngine (v1.1.0) supports the following features:

- Different techniques to divide the map into rooms. Only two techniques are implemented: equal division and tree division. Equal division divides the map into a grid then selects room from this grid, while tree division divide the whole map along the longest dimension till reach the required number of rooms.
- Define different tiles and define their maximum count.
- Define different neighborhoods in form of 2D matrix of 1's and 0's. 1's are the places to check while 0's otherwise.
- Define any number of cellular automata that the system will apply after each other.
- Specify where to apply the cellular automata. The system support two positions either applied on the whole map regarding of the room structures (useful for smoothing the whole map or generating game objects) or on the generated rooms (useful for designing dungeons).
- Connect/delete the generated islands after applying each

*"ProcEngine is an open source procedural map generation engine that allow the user to select from bunch of different generating algorithms and tune them."*

cellular automata.

- Cellular automata rules can have multiple conditions and replacing values.

The engine allows the users to modify the underlying generator through the following functions:

- **procengine.initialize(data):** to initialize the system with your rules.
- **procengine.generateMap():** to generate a level (you have to call initialize beforehand).
- **procengine.toString():** to get a string that shows the current data saved in the system.
- **procengine.testing.isDebugEnabled:** set to true to allow console printing after each step in the system.

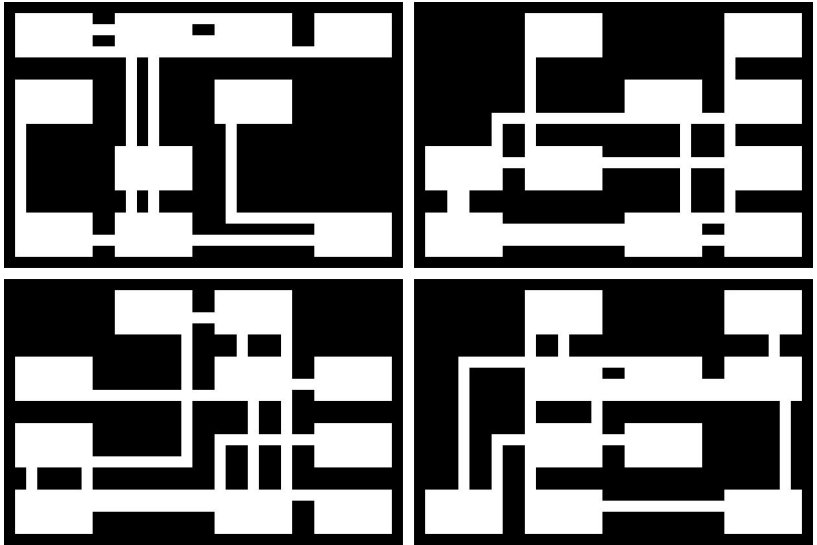
In order to use the system you need to call **procengine.initialize(data)** function first then you can call **procengine.generateMap()** for as many as you want. Each time you get a new generated map. For more details about how to use the engine refer to github (<https://github.com/amidos2006/procengine>).

Here is a bunch of examples that shows the capabilities of the system. The first example is a very simple generator. The generator should generate a map of 36x24 with 10 rooms using equal division technique.

```
var data={
  "mapData":["36x24", "solid:empty"],
  "roomData":["equal:4x4:10", "empty:1"],
  "names":["empty:-1", "solid:-1"],
  "neighbourhoods":{"plus": "010,101,010"},
  "generationRules":[
    {"genData":["0", "map:-1", "connect:plus:1"], "rules":[]}
  ]
};
```



Here are four different generated maps from the previous data, where white is empty and black is solid:



The second example is more complicated where it generates a map of 36x24 with 5 rooms using tree division technique. Also, it uses three cellular automatas in the following order:

1. Generate the solid structure of the rooms.
2. Connect the rooms together all over the whole map.
3. Adds objects (1 player, 10 gold pieces (at most), and 15 enemies (at most)).

```
var data={  
  "mapData":["36x24","solid:empty"],  
  "roomData":["tree:8x8:5","empty:2|solid:1"],  
  "names":["empty:1","solid:1","player:1","gold:10","enemy:15"],  
  "neighbourhoods":{  
    "plus": "010,101,010",  
    "all": "111,111,111"  
  },  
}
```

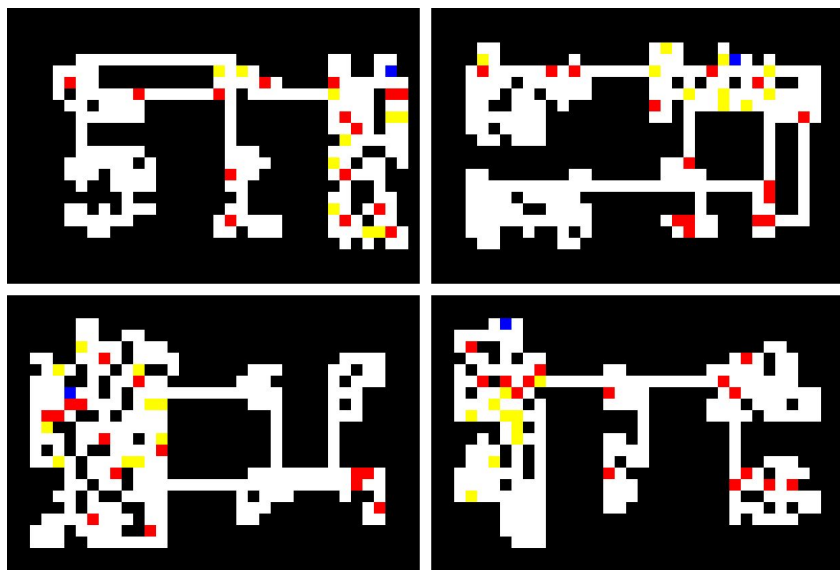
```

"generationRules":[
  {"genData":["3","room:-1","connect:plus:1"],
   "rules":["empty,all,or,solid>5,"solid:4 | empty:1"]},
  {"genData":["1","map:-1","connect:plus:1"],
   "rules":[]},
  {"genData":["1","room:-1","connect:plus:1"],

"rules":["empty,plus,or,empty>2,"player:1 | empty:8 | gold:2 | enemy:2"
]]
};

```

Here are four different maps generated from the previous data, where black is solid, white is empty, blue is player, red is enemy, yellow is gold:



# Gardening Games

By Max Kreminski

A lot of procgen-heavy games ask players to *explore*: to go out into the game world and actively seek out surprises amidst the procedurally generated landscape. Exploration of this kind tends to monopolize the player's attention; as you explore, you have to pay close attention to the terrain you're traversing, the landmarks you encounter, and the dangers that beset your path. You must keep your wits about you as you venture ever deeper into parts unknown.

In exploration games that feature large expanses of procedurally generated terrain, this often entails spending a whole lot of time looking at “samey”, repetitive content: the connective tissue that fills the gaps between sparsely distributed points of interest. With nothing to distinguish one massive flat expanse of desert from the next, the novelty of scale rapidly gives way to the tedium of picking your painstaking way across another hundred dunes.

*“Stories are fundamentally about change, and you can't witness change in anything or anyone besides yourself.”*


What happens once you finally do find something – a temple in the desert? In many exploration games, there's no real reason to ever visit the same place twice. The loop goes something like this: you travel until you discover an interesting place; investigate it as thoroughly as you like; take from it any resources you might want or need; and then keep pushing steadily onward, away from the clean-picked remains of your past.

This, as a format, is hostile to narrative. Stories are fundamentally about change, and you can't witness change in anything or anyone besides yourself unless you observe that thing or person repeatedly over a period of time. If you never encounter the same character twice, none of the characters will ever have any chance to undergo long-term change. This limits the stories that can be told about them to the scope of however much change they can undergo in the course of a single encounter.

\*

\*

\*



Gardening games are different. Where the exploration game requires its players to put in more effort if they want to encounter more surprising generated content, the gardening game keeps generating new content in the background – regardless of whether the player is paying attention to it or not – and brings any surprises it generates up to the player on its own.

The surprises of the garden are nothing as monumental as isolated temples in the desert. Instead, they are narrative surprises: surprises of cause and effect, of pushing on one small part of an interconnected system and watching the effects reverberate throughout the whole.

The player can use a variety of tools to exert influence on the garden, but the ultimate outcome is always shaped by forces entirely outside of the player's control. You can water certain flowers and plant certain seeds, but the weather doesn't always agree with your choices of which plants to favor. You can try to plant pink flowers over here and purple flowers over there, but don't be too surprised if – over the course of a few generations – the indiscriminate activity of pollinators erodes the sharp distinction between the two until it falls entirely away.

To play a gardening game is to become intimately familiar with the story of a bounded space as it changes over time. The player's attention remains fixed on a single, gradually evolving system; it is not scattered throughout a vast world whose individual parts are uniformly disconnected. To know why a garden looks the way it does today is to understand not only the histories of its individual parts, but also of the relationships between them, both past and present. In a garden, each individual tree becomes a character in an ongoing story, with a personal narrative arc all its own.

*“The player can use a variety of tools to exert influence on the garden, but the ultimate outcome is always shaped by forces entirely outside of the player's control.”*



\*

\*

\*





What games are gardening games? Neko atsume is a gardening game. Animal crossing is the quintessential gardening game. Stellaris, when played in certain non-expansionist ways, has something of the gardening game about it. Epitaph (<https://mkremins.itch.io/epitaph>), an idlegame I made for the fermi paradox jam, was initially conceived as – and largely remains – a gardening game.

Twitter bots, too, are garden-like in nature. You set up a generator and let it run, stopping by occasionally to search through its recent output for a harvest of surprising content. Although the underlying generative structure of a twitter bot is often painfully evident from only a small sample of its tweets, there is a great deal of pleasure to be had in seeing how the different elements of this structure sometimes conspire to produce funny or startling results.

Let a thousand gardening games bloom!



# Procedural Generation in Super-W-Hack!

By Ahmed Khalifa and Gabriella A. B. Barros

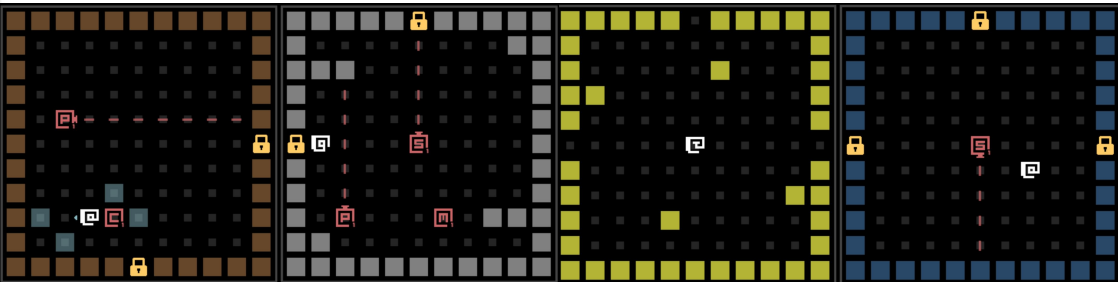
Super-W-Hack! is a synchronous (http://amidos-games.com/different-time-systems/) roguelike game where everything is automatically generated: levels, weapons, bosses, and sounds. It is a tribute to the roguelike genre, and is inspired by various games, such as NetHack, Super Crate Box, The Binding of Isaac, Spelunky, Sproggiwood, and more.

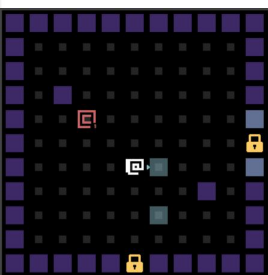
In Super-W-Hack!, the player explores five levels designed into a 2D map similar to those in The Binding of Isaac. To proceed, one needs to clear each room of enemies. After killing all the enemies, the player must accept a crate that will replace their current weapon with a new one.

In this game, weapons are represented as patterns. One will not see the actual weapon causing damage, but will see its effect on the map before choosing to trigger it. Our intentions behind weapon generation was to (hopefully) increase diversity, since one player may never get the same weapon twice, and encourage strategic planning.

Weapon generation starts with a pattern that is filled arbitrarily in a random size grid. Patterns can be centered on the player or appear in front of them. If placed in front, they may also be infinite patterns, which repeat themselves until they reach a wall or enemy. Additionally, playtesting showed us that most players had a hard time predicting patterns behaviors

*"After killing all the enemies, the player must accept a crate that will replace their current weapon with a new one."*





when they were too "noisy". Our solution was mirroring patterns in relation to the player.

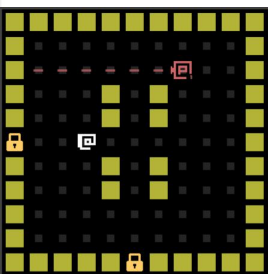
Finally, the weapon's name is generated using a combination of adjectives and nouns, and its sound effects are generated using sfxr. Both sounds and name are based on how good a given weapon is, which is calculated regarding how large is the area of attack and how protected is the player (can they attack from a distance?). A set of 100 weapons is generated at the beginning of the game, and sorted according to their power. Whenever the player gets a new weapon, one is selected and removed from the set, based on the level difficulty.

Levels in Super-W-Hack! are generated using multiple steps. First, the game chooses the dungeon name in the following format "The \#dungeonType of the \#adjective \#bossType}". \#dungeonType consists of 5 categories of dungeons which affects tile colors. \#adjective is a list of funny adjectives added to the \#bossType. \#bossType is a list of different objects, animals

and jobs.

After that, the game selects the map dimensions and generate a 2d maze of rooms using breadth first search, then assign a type for each room. Breadth first search is a search algorithm that, starting from a certain room, explores all non visited neighboring spaces and may transform it into a room, then repeats using the new rooms. visited. Possible room types are the starting room, an enemy room, an empty room, or a boss room (only on the fifth level of the dungeon).

Finally, as soon as the player enters, the game starts generating the room. %based on its room type. The first step is using cellular automata to generate the room structure. Cellular automata is a technique inspired by Conway's Game of Life. Each map tile have a probability to be either solid or empty based on the surrounding neighbors. After that, if the room type is an empty room, then produce a crate; if it is an enemy room, add enemies based on the level number and the



*"The first step is using cellular automata to generate the room structure."*

generated gun power.

Super-W-Hack! has 4 enemy types: (C)haser chases the player, (P)atrol moves horizontally or vertically and shoot laser if the player in front of it, (S)pinner rotates in the middle of the room and shoot laser if the player in front of it, and (M)iner moves randomly leaving a mine trail behind it. Enemies can attack each other if an enemy is in between the player and another enemy.

Bosses contain two or three behavior strategies, which can be either movement strategies (moves randomly, teleports, chases the player), attack strategies (leaves a mine on the floor, charges towards the player, shoots a single spot or a laser in front of it) or a special strategy (spawns enemies, heals itself). The generator selects strategies and creates the boss based on the \#bossType in the level name.

Super-W-Hack! was an ambitious project. We aimed

at implementing many interesting features in a short time span. Although we didn't have enough time for enhancing the game to its full potential, it received positive feedback for its nostalgic art style, generated weapons, fast paced gameplay and short respawn time. On the other hand, some found the game confusing, % a synchronous roguelike game with laser enemies was confusing as they expected the enemies to move after the player and not at the same time, and the current tutorial didn't clarify it enough. It was also a hard game, due to one-hit kills, enemies spawning too close to the player, among other reasons.

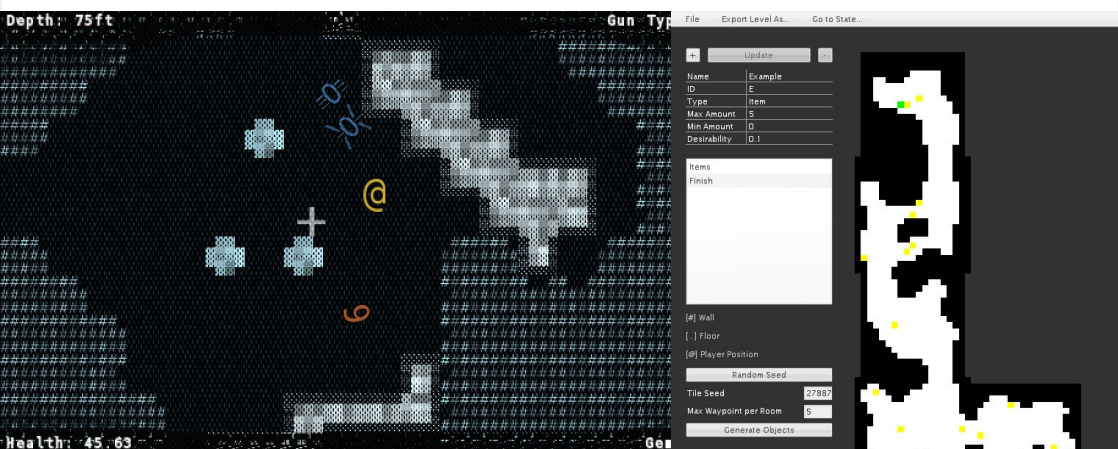
*"It was also a hard game, due to one-hit kills, enemies spawning too close to the player, among other reasons."*



# Caverns, Gems and Plenty of Text

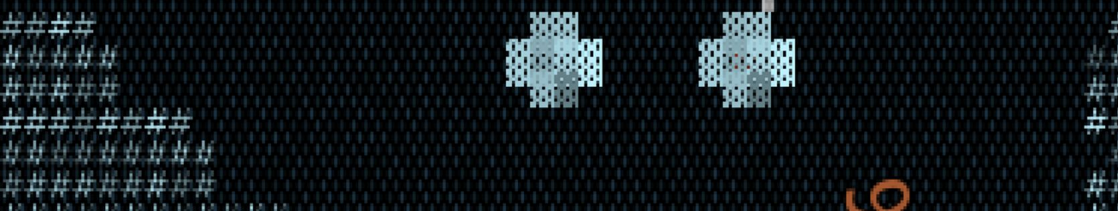
By Tim Stoddard

Back in September 2014, I decided that my final year thesis was going to involve procedural generation. I had enjoyed games with random yet functional levels such as Spelunky and Rogue Legacy that I wanted to explore the idea of creating levels from parameters. It has been two years and I'm still learning several approaches to procedural generation and even created my own to use in my current game, Gemstone Keeper, a twin-stick shooter roguelike with a heavy ASCII art style, procedural levels and gemstones!



It started with my thesis project, the Procedural Level Editor. This a program and library combined that let you generate multi-roomed procedural levels where you can adjust the parameters and preview at runtime. I wrote the library so the level generation can be separated into multiple parts, which also means the editor itself can preview each stage. Since graduating from University I've since updated the library and tool to include pathfinding, make overriding the parameters easier and even exporting individual levels in both text and images formats. I still use the Procedural Level Editor to generate the levels in Gemstone Keeper.





The Procedural Gemstones were something I created for PROCJAM 2016 back in February, as a means to display 3D gemstone graphics without creating and loading in models. I was inspired by methods to generate snowflakes by use of symmetry. By controlling the shape of the gemstone with the amount of lines of symmetry, I could create the vertices needed to create the gemstones I needed. I originally created the procedural gemstones in Unity, but Gemstone Keeper is written in C++ with SFML, so there was the fun task of writing a software 3D renderer. In the end I found creating procedural meshes to be a fun challenge that became very useful.



Since then I still find myself using procedural generation to solve problems. Recently I've been using Worley and Perlin Noise to add some ice and fire effects to the caverns, and using Markov Chains to generate the names for the gemstones. Ever since I started work on Gemstone Keeper, I've often enjoyed the challenge of making something from almost nothing, which is why almost all the graphics are made from a single text font file. When the object gets more challenging, the process to creating that object gets more creative.



# Stop Worrying And Love Constraint Solvers

By Martin Černý

A few months ago, I read a nice post by Kate Compton on creating generators (<http://tinyurl.com/seedscompton>). However one thing was bugging me – the post says that constraint solvers are not something you could easily use for PCG. Here, I want to convince you about the opposite – constraint solvers are great tools for PCG and implementing your own is easy.

So what are constraint solvers for? Let's say you have a dungeon map and want to decide what goes into individual rooms (enemies, loot ...). You also have some idea on how the dungeon should be composed – “There is always a healing item close to strong enemies”; “Total strength of all enemies is less than 200” or “No two adjacent rooms have the same content”. Or you develop an open-world game and you want to generate “bring me an item” side-quests using existing NPCs, so you need to choose an NPC as a quest giver, the item it wants and an NPC that has the item. You want the NPCs to be in reasonable distance from each other and the item must be something the quest-giver wants.

Both examples can be modelled as a bunch of variables (contents of the individual rooms / quest-giver, item and item-owner) where each variable is associated with a domain. A domain is simply list of possible values (enemies and loot / existing NPCs / item types) and every variable may have a different domain. Your design requirements then form constraints that say what combinations of values are OK. Constraints can concern a single variable (“the quest giver must like the player”), a pair of variables (“quest giver does not have the item”) or even multiple variables. The solution is then an assignment of the variables from their respective domains that satisfies all constraints. This forms a constraint satisfaction problem [1] which is solved by a constraint solver.

The nice part here is that you don't have to know how to find what

---

[1] Some of the terminology I use in this article may seem arcane, but it is used because of its Googleability.

“....constraint solvers are great tools for PCG and implementing your own is easy.”



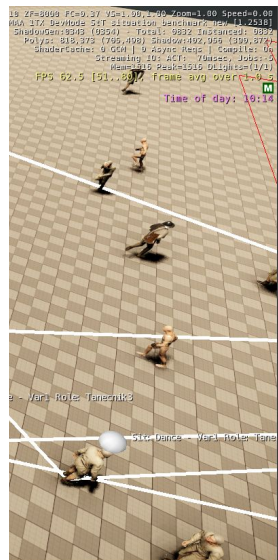
you are looking for, you only need to be able to recognize a valid result when you have found it.

So how do we implement a simple constraint solver for our generator? We combine two things: search (try all possible combinations) and inference (quickly eliminate obviously wrong possibilities). The search part (also called backtracking) goes like this:

1. Start: no variables are assigned a value, choose the 1st variable as currentVariable
2. Repeat [2]
  - a. If all variables up to currentVariable are assigned values that satisfy all constraints, move to the next variable (currentVariable++)
  - i. If there are no more unassigned variables, current assignment is a solution
  - b. Else assign next value to currentVariable.
  - c. If all values for currentVariable have been tried, unassign currentVariable and return to previous variable (currentVariable--). This is called “backtrack”.
  - i. If all values for the 1st variable have been exhausted, there is no solution.

Once you implement this you can add inference techniques, until the generator is fast enough. The beginner’s menu consists of:

- Node consistency: Before the search, check all constraints that concern only one variable and remove the failing values once and for all.
- Forward checking: After moving to a new variable in 1.a, scan the domains of the remaining unassigned variables (one at a time) and remove values that do not satisfy constraints involving the already assigned variables. Note



[2] Some sources describe the algorithm in recursive form. The forms are equivalent.



*“These three weird tricks are sufficient to solve small problems (as in the sidequests example) in microseconds!”*

that you have to remember which values were removed in this step, because they need to be returned on backtrack. Bitmasks are an efficient way of storing which values should not be tried.

- Backjumping: Upon backtrack, you can safely skip multiple variables back as long as the skipped variables are not involved in a constraint with the variable that caused the backtrack.

These three weird tricks are sufficient to solve small problems (as in the sidequests example) in microseconds! Adding more juice, (links below) can get you solutions for problems with few dozen variables (as in the dungeon generator) in milliseconds.

You should also not forget to randomize the order of variables in the domains prior to running the algorithm to get different results every run.

And that’s it! You have a solver!

Further reading:

- “How to build a constraint propagator in a weekend” by Ian Horswill and Leif Foged (includes C# code) <http://www.cs.northwestern.edu/~ian/GDCConstraintsHowTo.pdf>, also a related academic paper “Fast procedural level population with playability constraints” describing CSPs for filling in a dungeon. <https://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/viewFile/5466/5691>
- My work with CSPs for Kingdom Come: Deliverance described in detail in Chapter 6 of my thesis (C source code available) <http://popelka.ms.mff.cuni.cz/~cerny/thesis/> or a more condensed version in an academic paper: <http://www.aaai.org/ocs/index.php/AIIDE/AIIDE14/paper/view/8995>

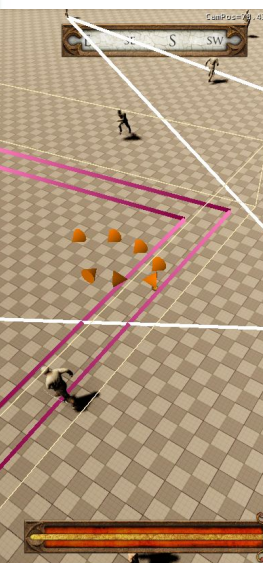


Figure: Debugging CSPs in Kingdom Come:Deliverance (finding tuples of NPCs for short events).

# Three Lenses For Generation

By Jo Mazeika

These are three different lenses to use when looking at your shiny new procedural generation system. These are not intended to be the only or best ways of thinking about a system, but are things that might be useful to keep in mind, regardless of the system's domain or generation method.

## i. Ontology

To generate something, we need to know what it's composed of. Songs are made of chords, which are made of notes; stories are made of characters and actions (all of these are reductive). But when we generate something, there are always atomic units. We can't break a music note into parts, and characters have traits, but typically we don't see traits as having subtraits. Defining the ontology for the generator (or the set of all possible concepts that exist within the generator) is a critical part of building the generator, since nothing that

is outside of the ontology can be output by the generator.

## ii. Mereology

With the set of things in the world in place, we need some way of describing how to combine them. Mereology, the study of parts and wholes, is the foundation for this. In order to make things from their constituent parts (events in a narrative, furniture in a room layout, organisms in a planet's ecosystem) we need to be able to describe how things are composed from which subparts. For any given artifact, there can be multiple ways of breaking it into component parts: a place setting is made of cutlery, plates, bowls and cups; or a place setting is made of a central dish, with some things to the left, some to the right and some above. This framework gives us a way of not only describing how our generated objects are comprised of their parts, but also a way of describing the

*"Defining the ontology for the generator (or the set of all possible concepts that exist within the generator) is a critical part of building the generator."*

set of things that can become parts of other things.

### iii. Semiotics

Semiotics isn't a new lens, at least in academia. It's the field of signs and symbols (the field that lets us say that this is not a pipe, but just a visual representation of one). It's easy to run right down the rabbit hole of saying that no things are ever generated, only representations of things. But that's not useful if you aren't concerned with the philosophical implications and are more concerned with making stuff that makes stuff. Where semiotics comes in is as follows: humans are very good at pattern matching and meaning making (thank you evolution). It's hard to look at :) without seeing the smile. It's hard to not find contexts and connections between any sort of generated materials.

favorite example of unfortunate implications here) but also allows the designer to leverage the power of useful symbols in context. Often this will involve an extra layer of design on top of the main generator, or careful planning on the ontology/mereology level.

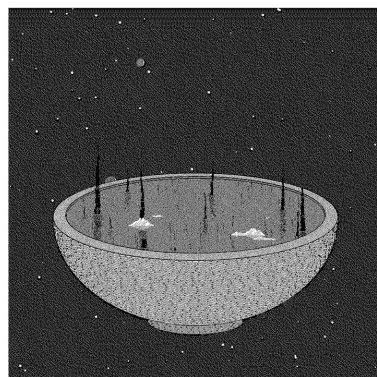
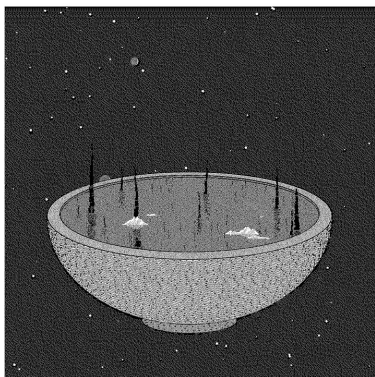
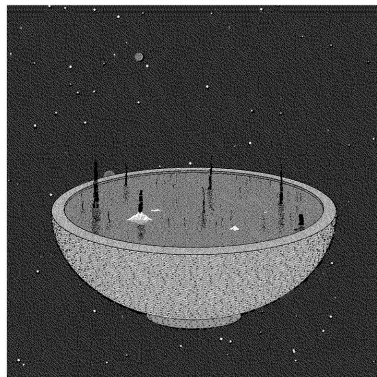
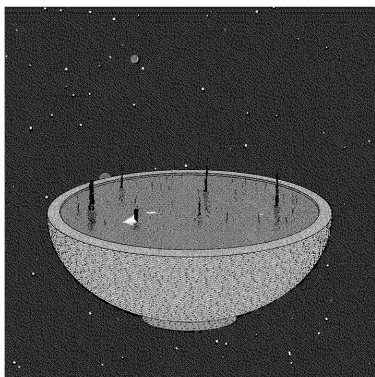
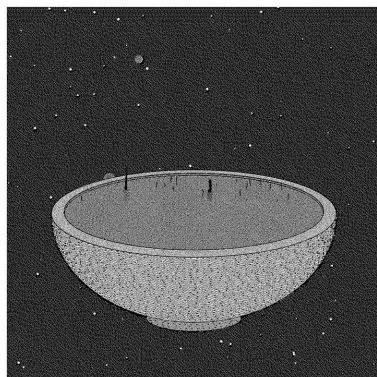
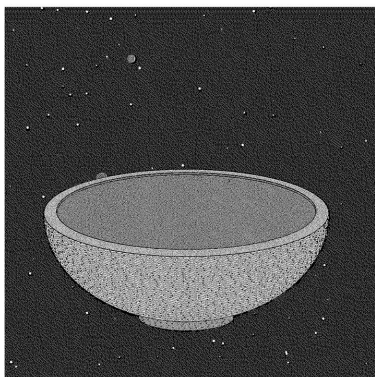
*"It's hard to not find contexts and connections between any sort of generated materials."*

Thinking about the semiotics of a system allows the designer to not only avoid unfortunate consequences of symbol combinations (insert your



# Mirror Lake

*By Kate Rose Pipkin*





# Tips for Terrain: Define Your Function in World Space

*By Rune Skovbo Johansen*



Judging from activity in the PCG community, procedural terrains is one of the most popular forms of procedural generation. There's no denying that to a lot of us, creating a terrain that you can emerge yourself in and explore is very appealing.

I've given this a go a few times myself, and I want to pass on a nice tip for working with terrain functions that has helped making things easier for me.

I'll skip the basics and I'll assume you've gotten a simple bumpy terrain up and running based on a noise function such as Perlin noise, Simplex noise or similar.

## **The tyranny of ranges**

It's likely that your framework requires inputs or outputs to be in certain ranges. For example, for a heightfield that accepts height values between 0 and 1, you might populate your height data like this:



```
for (int i = 0; i < resolution; i++) {  
    for (int j = 0; j < resolution; j++) {  
        //Pass array index co-ord i,j to the terrain func  
        data[i, j] = TerrainFunction(i, j);  
    }  
}
```

This easily leads you to define your functions such that they match those ranges. Your terrain function might look like this:

```
// Function takes x and z index into terrain data and returns height  
// in 0-1 range.  
float TerrainFunction(int x, int z) {  
    // Base height (0.4) is a bit lower than middle.  
    // Make each noise bump average to being 50 vertices wide  
    // and 10% high (+/-).  
    var heightValue = 0.4 + Noise(x / 50.0, y / 50.0) * 0.1;  
}
```

While this can work fine for a while, it creates friction down the line.

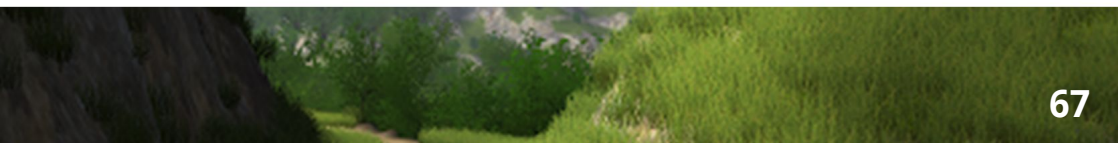
One of the first things you'll do to make your terrain more interesting might be to add together multiple noise functions with different scales. And at one point they might exceed the 0-1 range that the terrain accepts. Now you have to scale the output of all your noise functions down to compensate. If you have any calculations that take slopes into account, those need to be adjusted as well.


The same problem might occur if you find out you want the terrain to be more or less detailed, or if you decide to make the overall terrain area coverage smaller or larger. All the values in your function are also rather arbitrary, which makes them harder to visualize.

### Using world space units

The solution to all this is to not let your terrain functions depend on arbitrary ranges but define them in world space. Just define everything in meters or feet, or whatever unit you use in your world.

*“One of the first things you'll do to make your terrain more interesting might be to add together multiple noise functions with different scales.”*





```
// Function takes x and z coordinates in meters and returns
// height in meters.
float TerrainFunction(float x, float z) {
    // Base height is at 10 meters above 0 (sea level).
    // Make each noise bump average to being 40 meters wide
    // and 20 meters high (+/-).
    var height = 10.0 + Noise(x / 40.0, y / 40.0) * 20.0;
}
```

### Defining terrain bounds

To be able to do things this way you need to define your world space terrain bounds. These are just the bounds in world space your terrain was already taking up. You can derive those bounds from the existing size of your terrain, or you can define the bounds first and scale your terrain to fit. Either way you'll end up having bounds values you can make use of. For example like this (assuming y axis is upwards):

```
var minX = 0.0;
var maxX = 1000.0;
var minZ = 0.0;
var maxZ = 1000.0;
var minHeight = -20.0;
var maxHeight = 40.0;
```

*“These are just the bounds in world space your terrain was already taking up.”*

### Converting coordinates from array indices to world space

You can convert from array indices to world space coordinates with these conversions:

```
for (int i = 0; i < resolution; i++) {
    var x = minX + (maxX - minX) * i / resolution;
    for (int j = 0; j < resolution; j++) {
        var z = minZ + (maxZ - minZ) * j / resolution;
        // Pass world coordinate x,z to the terrain function...
        height = TerrainFunction(x, z);
    }
}
```

### Converting heights from world space to 0-1 range

And as a very final step, you can scale your world space height into a 0-1 range using:

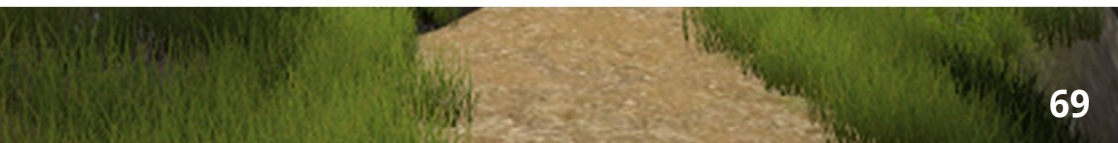


```
for (int i = 0; i < resolution; i++) {  
    var x = minX + (maxX - minX) * i / resolution;  
    for (int j = 0; j < resolution; j++) {  
        var z = minZ + (maxZ - minZ) * j / resolution;  
        // Pass world co-ord x,z to your terrain func  
        height = TerrainFunction(x, z);  
        heightValue = (height - minHeight) / (maxHeight -  
minHeight);  
        data[i, j] = heightValue;  
    }  
}
```

Now that your data format and your terrain function is completely uncoupled, you can change the terrain resolution, or the area the terrain covers, without having to change your functions. And you can mess about with your functions in meters (or whatever) without thinking about fitting them into a specific range. If they go out of range, just increase the range accordingly in your defined bounds, and everything is well again.

You can read more about procedural generation on Rune's blog at <http://blog.runevision.com>

*“And you can  
mess about with  
your functions in  
meters (or  
whatever)  
without thinking  
about fitting  
them into a  
specific range.”*





# Be Less Random with rand()

By Aidan Dodds

@Aidan\_Dodds

## Introduction

It should come as no real surprise that most procedural content generation (PCG) systems are underpinned by a good random number generator. Most programming languages provide a means to generate random numbers; traditionally via a `rand()` function. This function typically generates uniform random numbers, which is to say, any number has the same likelihood of being returned as any other. If you are looking to implement your own then a nice starting point may be the Xorshift PRNG.

Procedural content however is less about randomness, and rather more about building upon sources of randomness to create unique and artistic content. In this article, we will look at `rand()` and see how it can be extended into something more versatile.

I feel a quick disclaimer is in order however; I am a programmer, not a statistician, so while the following techniques have served me well for my PCG needs, there is a good chance my math or terminology is wrong...

## Starting point

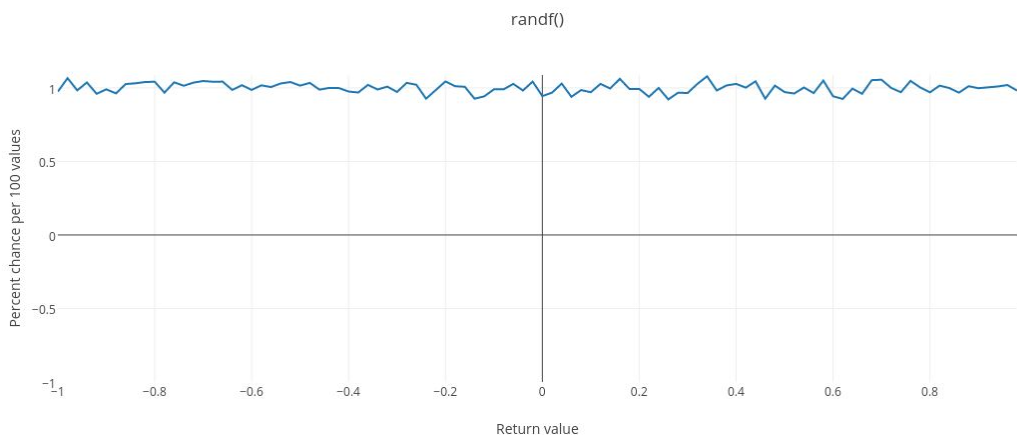
As a starting point, let's assume that we have a function `randf()` that returns a uniform number in the range  $[-1, +1]$ . Such a function may already be provided by your language but is generally trivial to implement, for instance:

*"Procedural content however is less about randomness, and rather more about building upon sources of randomness to create unique and artistic content."*

```

...
function randf()
  # where rand() returns a random unsigned integer
  return 1.0f - float(rand() % 4096) / 2048;
end
...

```



Uniform distributions however are often not the best fit artistically for a game or PCG system. It can be very useful to have control over the probability of our generated values. So let's look at some alternatives to the uniform distribution, and how to produce them (in pseudo code form):

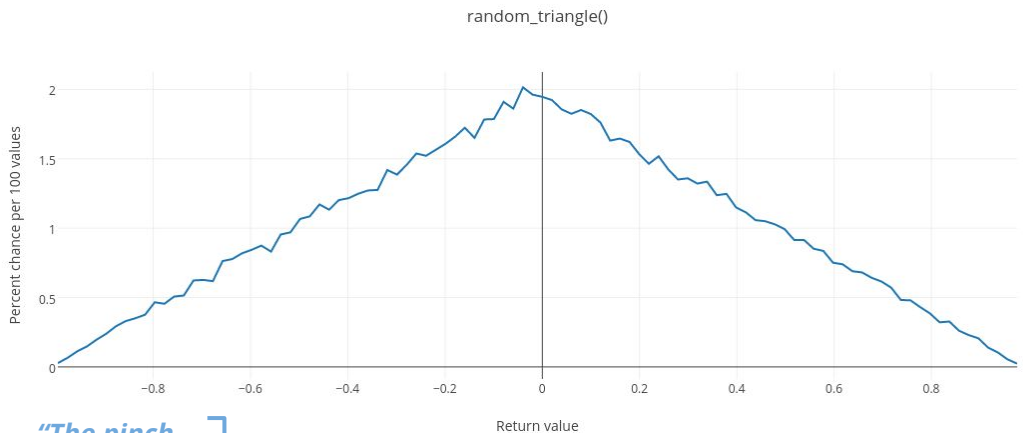
## 1D distributions

### Triangular distribution:

By taking the average of two random numbers it can be shown that there is a much stronger chance of a value near 0.0 being produced than that of 1.0 or -1.0.

Such a distribution can be useful when you want to add a some variance to data with a few large variations and substantially more small variations.

```
```\nfunction rand_triangle()\n  return (randf()+randf()) / 2.0\nend\n```\n
```



*“The pinch distribution as I call it (because i do not know the correct term) is somewhat like an extreme version of the triangle distribution.”*

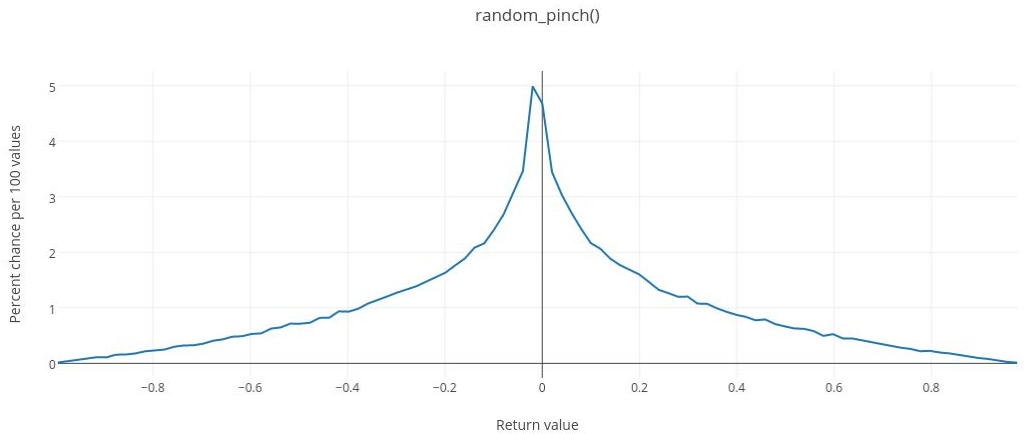
### Pinch distribution

The pinch distribution as I call it (because i do not know the correct term) is somewhat like an extreme version of the triangle distribution. values near 0.0 are very probable where as it is rare that values near 1.0 and -1.0 will be returned. This distribution has great results when used for adding a little variation to firing lines for example. I have also had nice results using this distribution to effect the direction of each element in a particle system.

```

...
function rand_pinch()
  return randf() * abs(randf())
end
...

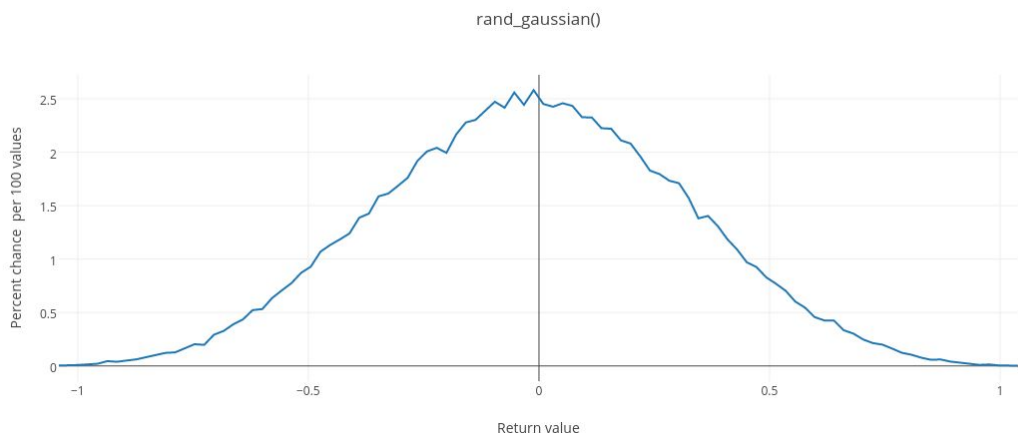
```



### Gaussian distribution

A Gaussian distribution (or normal distribution as its also known) can be constructed, and is very distinctive with its bell like appearance. Interestingly the higher to number of rounds the better the approximation becomes. This distribution can be nice when you want a good range of values with a few larger outliers. This could make a nice basis for generating good looking star systems.





## 2D and 3D distributions

### Random 2D vector in a circle

When writing procedural generation systems it is often desirable to be able to generate a 2D or 3D vector that falls uniformly within a circle or sphere. That is to say the vectors direction is random, and its magnitude ranges from  $(0.0, 1.f]$ . This can be useful for applications such as random sampling around a point, making random walks and stochastic approximations like ambient occlusion.

```

...
function rand_circle()
    float x = 0.0, y = 0.0
    while (True)
        x = randf()
        y = randf()
        if ((x*x + y*y) <= 1.0)
            return (x, y)
        end
    end
end
...

```

### Random unit 2D vector

Generating a good random unit vector (vector with length 1.0) can be a little more trick then it first seems. The most obvious solution would be to randomize the x, y and z components and then normalize the vector; which however produces a less then ideal vector since it will be biased towards diagonals.

We can generate an unbiased vector by starting with our `rand_circle()` function before normalizing it.

```

...
function rand_unit_vector()
    return vector_normalize(rand_circle())
end
...

```

### Random 3D vectors in a sphere

The same approach we took for generating 2D vectors can easily be

*“...which  
however  
produces a less  
then ideal vector  
since it will be  
biased towards  
diagonals.”*

extended to three dimensions as follows.

```
```\nfunction rand_sphere()\n  float x = 0.0, y = 0.0, z = 0.0\n  while (True)\n    x = randf()\n    y = randf()\n    z = randf()\n    if ((x*x + y*y + z*z) <= 1.0)\n      return (x, y, z)\n    end\n  end\nend\n```\n
```

Like we did before, if we normalize this vector then we can produce an unbiased unit 3D vector.

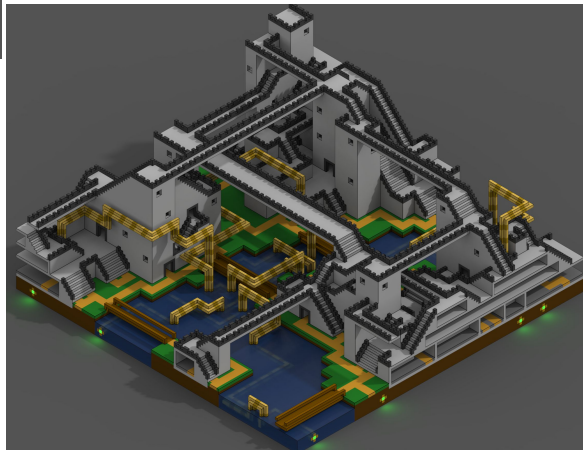
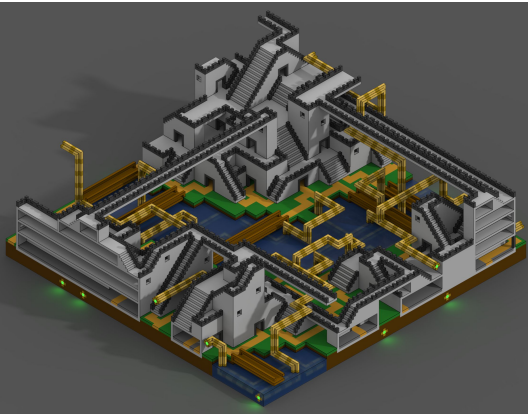
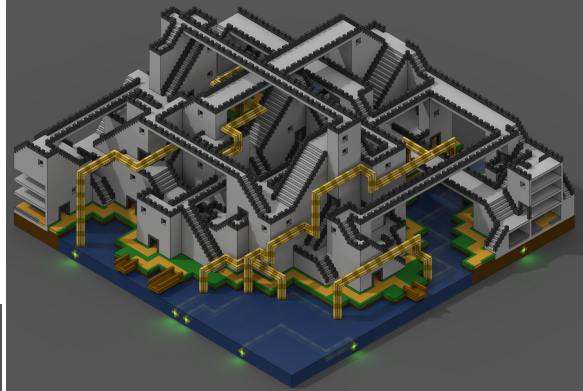
### **In closing**

A few relatively simple techniques to generate more interesting random numbers have been presented. Where and how they are applied is still firmly where the artistic element of procedural generation lies. Like an artist however, its always good to have more brushes to paint with.

# Voxel Models

By Ex Utumno

Voxel models generated  
with the wave function  
collapse algorithm  
<http://github.com/mxgmn/WaveFunctionCollapse>





# Challenges and Experiences in Generalized Level and Content Generation

By Afshin Mobramaein, UC Santa Cruz

In the world of procedural content generation (PCG) there have been a wide myriad of systems and algorithms that create new and interesting content such as levels, assets and even complete games. But one of the limitations of these generators is the one of domain specificity. Level generators for instance, focus on either one very specific type of games (i.e Super Mario Bros), or assets (terrains, trees, textures). As of recently there has been an increased interest on generalized content generation, an example of this is the GVGA [1] competition on level generation. Removing the constraints of domain specificity might lead into a new series of generators that can create content that might adapt to innovative ideas of gameplay.

One field in which generalized content generation might come in handy is the one of automated game design (AGD) with systems capable of creating new game rules and mechanics. Building larger-scale AGD systems implies going beyond rules and mechanics into the generation of gameplay spaces and assets that “make sense” with the new types of game rules and mechanics that are being generated. Since AGD systems create the context for level and asset generation, generators should adapt to create compelling content regardless of whatever context is thrown at them. This invariably leads to a series of very interesting research questions and design considerations.

The first question to arise, is how do we set up an architecture to create context independent generators for levels in AGD systems? Assuming we already have a rule generation mechanism in place, how do we generate levels that best fit the context generated by our rule generator? An initial suggestion would be to integrate an intermediate layer that can associate specific patterns in game rules to a notion of genre (as different types of games yield different level geometry) in game, and from there to choose an appropriate level generation mechanism that better suits the understood context from our rule generation process. This notion is explored by Zook and Riedl [2] in their paper “AI as a Game Producer” in which creative direction is given by a producer layer that has knowledge about genre and can lead a series of generative systems to create a game.

But how do we map rules to a notion of genre? From this point, we could imagine using several different approaches: For instance we could build a ruleset that maps certain elements of game rules (player affordances, goals, camera relationships, world physics) into

*“Since AGD systems create the context for level and asset generation, generators should adapt to create compelling content regardless of whatever context is thrown at them.”*

a game archetype genre that fits the best. This can be seen as a starting point, and one that will require a large knowledge engineering effort. On the other hand, we could picture using a data-driven approach in which we can cluster different types of games from a standardized corpus (think VGDL or PuzzleScript) to learn a notion of genre, and then apply a classification model for new observations generated by our system.

Another important question that arises is the one of which generative method to use after a type of context has been determined. One strategy that could work is to leverage the power of the wealth of great generative systems that the PCG community has used and to choose a generator that fits the type of game that fits the generated rules better. While this sounds like a very viable option, it also implies that a standardized knowledge representation of what a game level looks like. A more modern approach could also involve using deep learning techniques, such as generative models with a corpus of video game levels such as the VGLC by Summerville et. al [3] as its training set. Finally, an approach such as the one explored by Sorenson and Pasquier [4] in their paper “Towards a Generic Framework for Automated Video Game Level Creation” that relies in breaking down levels into building blocks called “design elements” from different types of games can be used. A concern here is the one explored by our ruleset approach for our genre mapping, is the one of knowledge engineering to encode a large set of different design element families.

*“...that relies in breaking down levels into building blocks called “design elements” from different types of games can be used.”*

The questions above are some of the emerging issues surrounding generalized level and content creation, and there are clearly more questions that are as interesting such as the ones involving evaluating the quality of generated artifacts. But for now, this seems to be a promising area in PCG research and the future seems to hold some interesting systems being developed such as the ones that competed in the GVGAI content generation track this year. Hopefully, we will see a large wealth of great generalized level generators in the near future, and with that a myriad of lessons that we can learn from them.

## References

- [1] Perez-Liebana D et al. “General Video Game AI:Competition, Challenges and Opportunities”, 2016, *In AAAI 2016 Proceedings*.
- [2] Zook A, Riedl, M, “AI as Game Producer”, 2013, *In CIG (Computational Intelligence in Games) 2013 Proceedings*.
- [3] Summerville A et al, The VGLC: The Video Game Level Corpus ,2016, *In Proceedings of FDIGRA 2016 7th Workshop on Procedural Content Generation*
- [4] Sorenson N, Pasquier P, Towards a Generic Framework for Automated Video Game Level Creation, 2010, *In International Conference on Evolutionary Computation in Games, EvoGame Proceedings*.

# Lucidity

*By Scott Redrup*

Watch online at <http://tinyurl.com/seedsLucidity>

## **Who I am:**

I am Scott Redrup and I have spent the past three years completing a Bsc (hons) degree in Computing and Game Development at Plymouth University. In my final year I completed a substantial project that explored procedural level design, and created a game called Lucidity.

## **What I did**

The project had two components, a research phase and an implementation stage with a final demonstration. I analysed current work within the fields of procedural generation, level design, procedural level design and game design patterns. This allowed me to identify current issues and propose a solution. My final product is a 3D dungeon crawler with seven different level objectives that demonstrate how game design patterns can be used with procedural generation to create interesting levels.

## **Research**

I had little knowledge about the current work within level design so I spent a lot of time reading the works of Togelius, Dahlskog, Bjork amongst other notable game AI researchers. Significant time was invested in analysing design patterns, starting with Alexanders pattern language compared against the gang of fours approach to software engineering and finally Bjorks collection of patterns. This led to interesting discussions about how patterns can be identified, classified and categorised. All worthy research projects in their own respects.

## **Implementation**

Levels were generated by first creating the basic level structure. Levels feature two heights of elevation, the ground and mountains layers, both of which are flat. Players can access the mountain layer via stairs. The level is built by carving the ground into mountains by using a random walk algorithm and then searching for a suitable location to place stairs.





The level is then split into a grid of 5 x 5 chunks. Each chunk represents an area of space within the level and contains information for how the area should look i.e what scenery is generated and its arrangement.

The third step is to tailor the basic level in order to create the 7 different level types. To create 'Arena Battles' the levels had to be searched for spaces to place an arena. Whereas for levels like 'Lots O Enemies', enemy spawn rates had to be increased.

Finally basic gameplay elements had to be added including a win / loss condition e.g being killed, as well as a menu, tutorial screens etc.

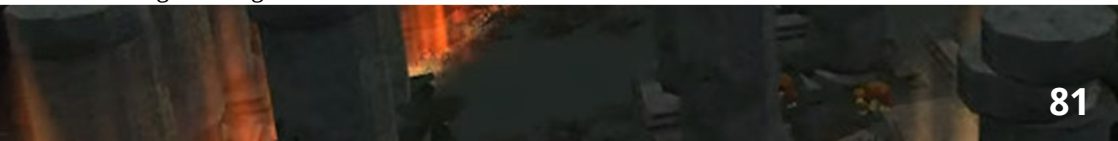
### Evaluation

Three main issues were encountered with my solution:

- **Code Structure:** Procedural projects are really fun and it is easy to make something quickly, without paying attention to the structure of the back end. I fell for this trap and by the end of the project trying to add or remove features proved impossible.
- **Performance:** I relied on the Unity Asset Store to make a game that looked good. With a limited range of free assets available, Lucidity ended up with lots of poorly optimised assets cause significant lag on most low-mid ranged machines. Levels with specific spawning requirements such as Arena Battles caused 1-3 seconds of initial lag.
- **Difficulty:** I envisioned levels were to be assessed on how easily the difficulty rating could be adjusted. This would link to procedural enemies that would vary in health, damage, speed etc. Which would be set by a difficulty class, due to the aforementioned code structure issues as well as a lack of time, I failed to implement the feature.

### Discussion and Going Forward

I'd be interested to see my approach applied within a large scale application as I'm unsure if the levels would remain interesting or become repetitive. I'm also unsure if the approach is optimised to handle large scale generation.







*"So ensure that you go into the project knowing what you want to achieve, use version control and commit often!"*

I managed to achieve an illusion of variety in *Lucidity* by using a forest themed environment and vegetation appeared to hide repeating patterns effectively. In an urban environment this would I predict this would be more noticeable.

One thing I recognised with most of the research I carried out was that a lot of the work is still only concerned with simple game genres such as platformers and dungeon crawlers. I'd want to see my work applied to a more complex genre.

### **Conclusion**

To conclude, 80% of this project was spent thinking that I'd make a procedural thing that did nothing, and there were certainly a few moments where I'd hit play in Unity and a small change had messed up the whole generation. So ensure that you go into the project knowing what you want to achieve, use version control and commit often! BUT! Procedural projects are scary, yet definitely give it a go and explore! There is something strangely satisfying about generating unique complex levels at the click of a button!



# Ultima Ratio Regum

By Mark Johnson

This year in the ongoing development of *Ultima Ratio Regum*, a ten-year experimental roguelike project focused on the procedural generation of culture and cultural behaviours, my focus has been almost entirely on *people*. The world has been notoriously devoid of human life for several years despite the tremendous social, religious and political detail that has gone into the worldbuilding, and it was finally time – with all these foundational elements in place – to change that.

Firstly, what should they look like? I wound up creating an interwoven two part model of biological and cultural NPC elements. On the biological front, we have a range of variations: different genetic groups have different randomly-selected shapes of eyes, chins, necks, ears, noses, and so forth, alongside different colours for their hair, and their eyes. Skin tone of course varies with how close to the equator a particular person's family originally hail from, with an appropriate range of variation from the darkest black to the palest white. I then combined these with cultural elements, which take two distinct forms: cultural elements that are applied to an NPC's face (the only part of their "body" you can see in-game), and those applied to the items (clothing, weapons, etc) that a character happens to carry with them. On the faces of NPCs we find a massive range of hairstyles for both women and men which vary with culture, along with sets of distinctive cultural practices: scarification, tattooing, specific kinds of jewellery, turbans, paint markings, and many others.

*"...project focused on the procedural generation of culture and cultural behaviours, my focus has been almost entirely on people."*



This was then joined by clothing styles, for which I found myself building a rather detailed procedural clothing style generator. Clothing styles can have shirts and trousers, waistcoats and skirts, dresses, or togas, or anything in-between, with additional variation



in style and appearance determined by the overall aesthetic preferences of the nation in question for certain shapes, certain colours, and so forth. Styles are distinctive either to entire cultures, or to niche demographics within a culture, such as the religious clergy, or soldiers. Each clothing style then breaks down into multiple tiers, helping the player identify the status of an unknown NPC and adding far greater variation to this part of the game visuals.



This therefore allowed for the interesting intersection of biological and cultural traits, and the ability for the player to play detective. Consider an empire from an equatorial region – the player is used to encountering characters with a dark skin-tone wearing a certain set of clothing and jewellery. At some point, however, the player happens to bump into a pale-skinned character with a different hair colour, who nevertheless possesses the same clothing styles (so biological difference, cultural similarity). Does this person represent a conquered colony? A trader trying to fit in? A slave or servant? Or something else? Nothing of this sort is ever explicitly told to the player, and so the player must instead rely on their knowledge of that particular generated world in order to draw conclusions based on their physical appearance, their clothes and any facial cultural traits, as well as their actions and patterns of speech, which brings us to our latter point – what NPCs actually do.

*“...so the player must instead rely on their knowledge of that particular generated world in order to draw conclusions...”*

Developing NPC behaviours means how they spend their day, and how they talk to the player. The NPCs in *URR* now range from mercenaries to priests, guards to merchants, farmers to inquisitors, and arena fighters to servants and eunuchs. Each NPC class spawns and lives in a different part of the map and has a very different set of rules for their average everyday behaviours – take, for example,





this screenshot of priests and worshipers (standard humans, shown with an “h”) going about their day.

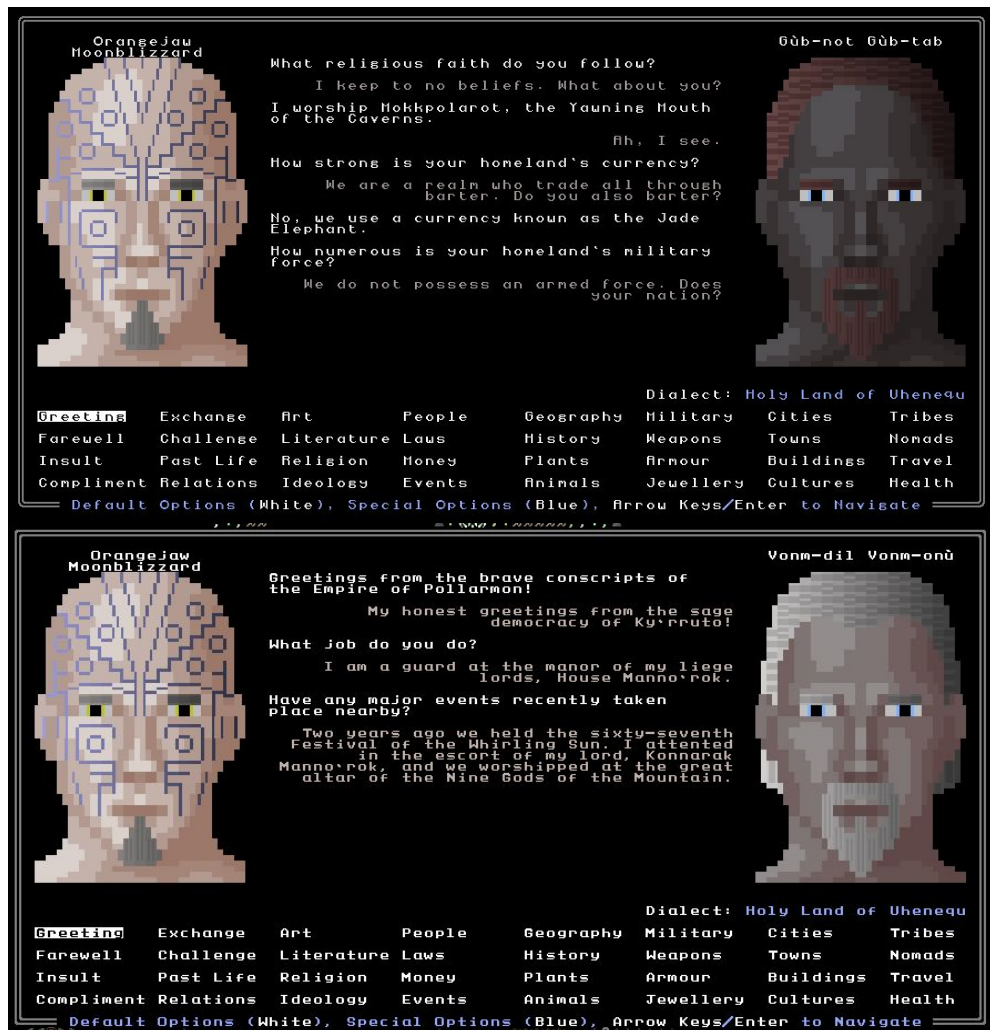


These highly active NPCs are then married with a pretty unusual speech system. Joining us now in this last part of 2016’s *URR* journey will be Orangejaw Moonblizzard, my profoundly procedurally-generated and facially-tattooed playtesting character who has travelled with me for over a month now – which is to say, I haven’t in this time needed to generate a new world to experiment with, and thereby expunge brave Orangejaw from existence. The goal was to create a speech system where the player could ask a tremendous range of questions without having to resort to programming it as a “chatbot”, to create realistic (or at least realistic-ish) human conversations, and to allow the player to uncover large volumes of information about the game world simply by speaking to its inhabitants. AS things stand now, I feel very





confident this objective is almost complete:



With all of these elements (almost) complete, URR now finds itself replete with a procedurally-generated cast of culturally-detailed characters, ready for the player to discover, watch, talk to, and perhaps find out crucial clues from...



# Techne

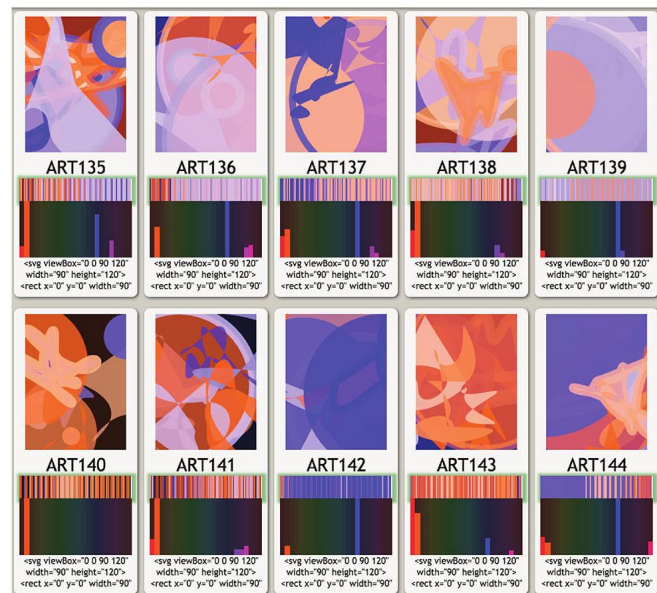
## *A commune for artbots*

A project by Kate Compton, Johnathan Pagnutti  
Jim Whitehead and many others contributors!

*Art bots are a lot of fun. Push a button and get some art!*

But real human artists don't work like that. Human artists gather in artist communities, and argue and learn and share new idea about art.

If we made a virtual art colony, would we get artbots that acted in ways that felt more human? Would they tell us interesting things about creativity? **Would they make good art?**



Techne works by using a meta-grammar to make art-generating grammars. We use **Tracery**, a text-generation library by Kate, and the art itself is made from layers of SVG graphics.

**Want to know more details?**  
Read an academic paper

"Do You Like This Art I Made You:  
Introducing Techne, A Creative

**Artbots have a lot of opinions about art.**  
**Their opinions are generated too!**

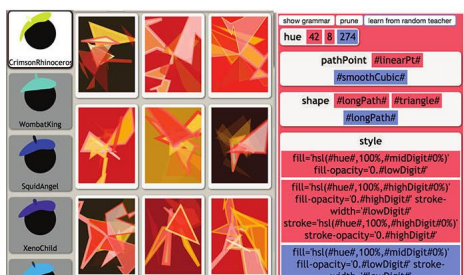
Some bots like pink, others like art that looks like cats or has sharp edges. If we can detect it algorithmically, a bot can have an opinion about it!





## Evolving some art

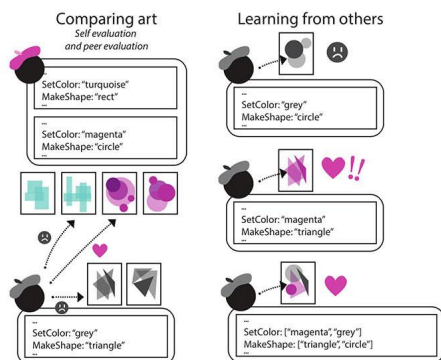
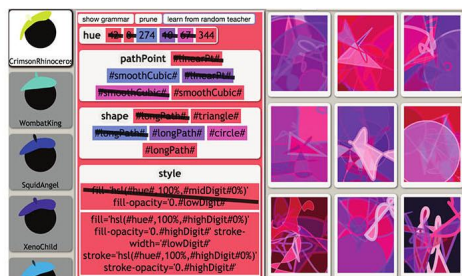
CrimsonRhinoCeros makes flame-colored spiky art, using their art-generation grammar (on the right)



But they can learn from a new teacher (another bot and their grammar). Now CrimsonRhinoCeros can use magenta, and knows how to make squiggles, too!

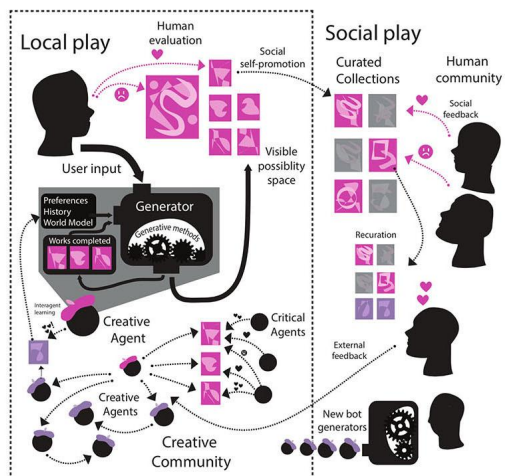


Sometimes forgetting old techniques is as important as learning new ones. We can also "prune" the generative grammar, forgetting the colors orange and yellow.



Bots can evaluate their art, and evaluate the work of others. The can become superfans of a different bot, or a self-loathing artist or a proud individualist who ignores the popular trends.

Techne is still changing and growing. We hope eventually it'll be a whole decentralized world where many bots (and maybe some humans) can come and play.





# Elision

By Isaac Karth

[procedural-generation.tumblr.com](http://procedural-generation.tumblr.com)

Intending to travel by road to Naissus, Virgil left Ulpiana. It was at least 80 miles.

He passes another milestone. Along the road are graves, and a cenotaph. An oxcart passes, loaded with grain. The road narrows here, an orchard wall encroaching on it. There a spring wells up, and around about it is a meadow.

\* \* \*

Intending to travel by road to Naissus, Virgil left Bononia (Moesia). It was at least 76 miles.

A cloud passes in front of the sun. As they go up from Bononia (Moesia), they see the ruined walls. A grove of Minerva is hard by the road, a grove of poplar trees. The sun beats down. Now the road is quieter. Not far from the road is a grave, on which is mounted a soldier standing by a horse. Who it is I do not know, but both horse and soldier were carved by

Praxiteles. Workers are raising the level of the road. This is a smooth road, by which many wagons were bringing wood to Naissus.

\* \* \*

From Ancona to Iader is a journey of about 107 miles when travelling by ship down the coast.

Out of the clouds bursts fire fast upon fire. Dubious days of blind darkness we wander on the deep, nights without a star. Then comes the creak of cables and the cries of seamen. Frequent flashes light the lurid air. All nature, big with instant ruin, frowned destruction. The oars are snapped. Piteous to see, it dashes on shoals and girdles with a sandbank. The helmsman is dashed away and rolled forward headlong.

Then was land at last seen to rise, discovering distant hills and sending up wreaths of smoke.

*"Out of the clouds bursts fire fast upon fire. Dubious days of blind darkness we wander on the deep, nights without a star."*



Within a long recess there lies a bay: an island shades it from the rolling sea and forms a port secure for ships to ride. Two towering crags, twin giants, guard the cove, and threat the skies. Betwixt two rows of rocks a sylvan scene appears above, and groves for ever green. Beneath a precipice that fronts the wave, with limpid springs inside, and many a seat of living marble, lies a sheltered cave. Ships within this happy harbor meet, the thin remainders of the scattered fleet. They lay their weary limbs still dripping on the sand.

imply relationships while not relying on the existence of any of the other phrases. The reader fills in the gaps left by the missing phrases.

Additionally, this technique let me borrow many of the phrases from Roman travel literature or Virgil's own poetry, lending another layer of structure, allusion, and meaning.

*"The generator picks a fraction of phrases from the list and joins them together."*

\* \* \*

For Virgil's Commonplace Book, which I generated for National Novel Generation Month 2015, I made use of elision, a literary trick I learned from Nick Montfort's 1K Story Generators.

Each kind of connection has a list of evocative sentences describing the journey. The generator picks a fraction of phrases from the list and joins them together. Many of the phrases are atmospheric and

# Is Self Similarity Too Similar

By Mark Bennett

A question. How does procedurally generated terrain compare to the real thing? Having had opportunities to travel and see a variety of landscapes, my conclusion is; not very well. Real terrain is very varied and often has distinctive features, which is what can make a particular terrain striking to the eye.

I have decided to use Outerra (<http://www.outerra.com/>) to compare real terrains with their corresponding fractal terrains as Outerra uses heightmaps of real terrain (Earth) to a specific resolution (30m) and interpolates between height points using fractal methods (described in more detail here <http://www.outerra.com/procedural/demo.html>), therefore allowing a direct comparison. This article is not a critique of Outerra which is an outstanding piece of software.

I have also performed some experiments using the midpoint displacement algorithm and shown the results below.

## Some Examples Of Real World Terrains

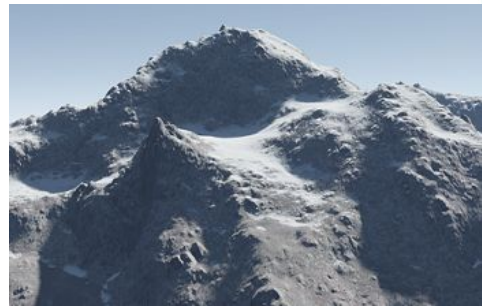


## Comparison of Real vs. Procedural Terrain

Below are images of real terrains, compared with their fractalised versions using Outerra from as close a viewpoint as possible. Some of the character of the original terrains is lost when fractal methods are used to interpolate between height points.



*Jungfrau (Switzerland)*

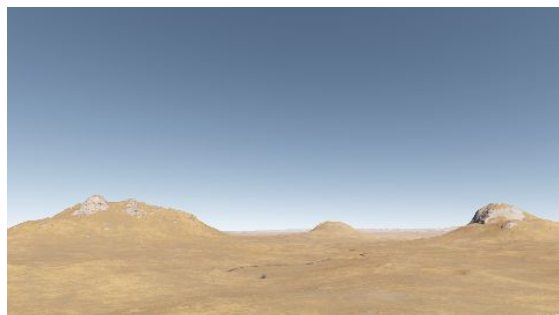


*Jungfrau (Outerra)*

The Jungfrau loses the beautiful Silberhorn to the right of the main summit.



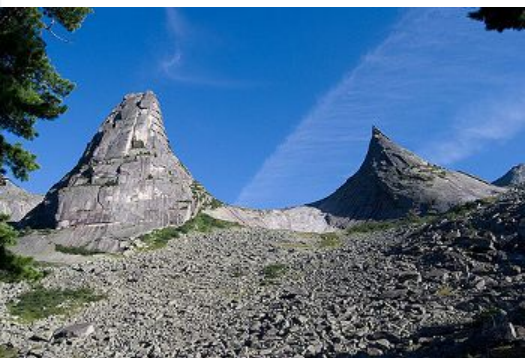
*Monument Valley*



*Monument Valley (Outerra)*

The distinctive towers which make Monument Valley in the USA such a big tourist attraction disappear when fractalized as can be seen above.





*Prabella Mountain*



*Prabella Mountain (Outerra)*

Parabella mountain in Russia has a very distinctive central ridge from which it gets its name. This also does not survive the fractal sledgehammer.



*El Capitan*



*El Capitan (Outerra)*

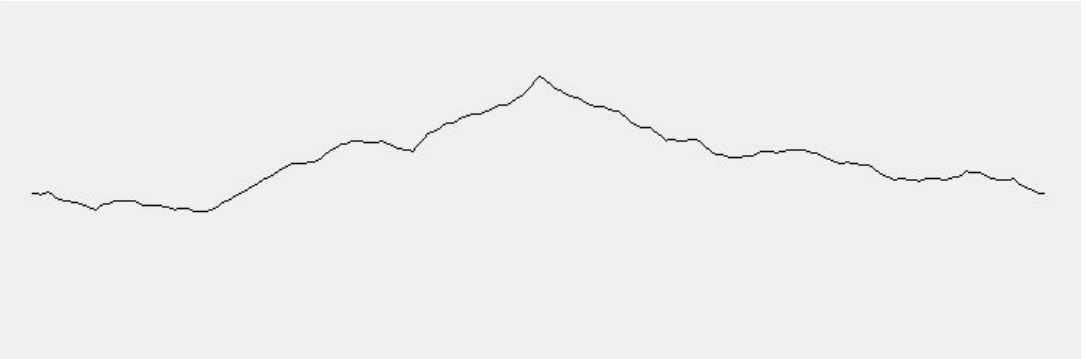
El Capitan, in the Yosemite Valley in California gains a series of jagged peaks on its summit which are not present on the original, by contract, the huge cliff face loses all of its features.

## Experiments

I implemented the simplest possible fractal algorithm, the Midpoint

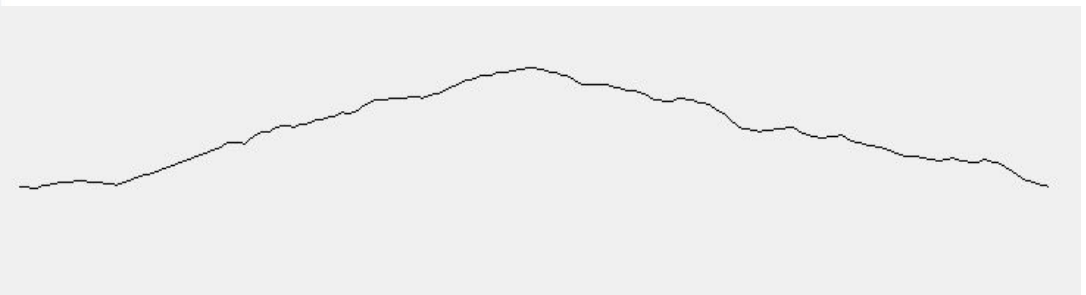


Displacement Algorithm and set about tweaking some of its parameters. All these experiments use the same seed for the random number generator. The Python code is available at: <https://github.com/MarkBennett12/MidPointDisplacementExperiments>



*Standard Midpoint Displacement*

The output of the unadorned algorithm.



*Random Lacunarity*

Here the lacunarity has been randomised (using a separate RNG) by only allowing a 50% chance of displacement at each iteration.



### *Random Midpoint Placement*

Here the position of the midpoint has been randomised (separate RNG again). This allows cliffs and scale variations to appear.



### *Random (Triangular) Midpoint Placement*

Randomised midpoint again but using the triangular random function to get more consistent output.

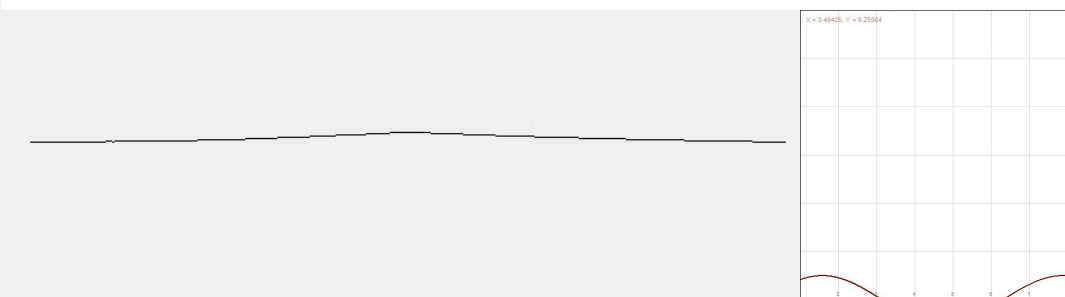


### *Random Midpoint and Random Lacunarity*

Both Lacunarity and midpoint randomised

Some additional experiments were performed using simple functions to modulate the amount of random displacement. Again, the same random seed is used each time. The output of the function is normalised to as close to 0 and 1 as possible and clamped so as not to go below 0. The output is on the left, the function shown below and the graph of the function shown on the right.

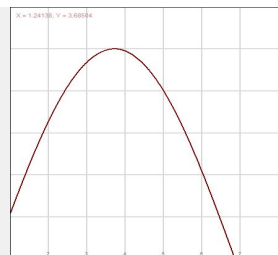
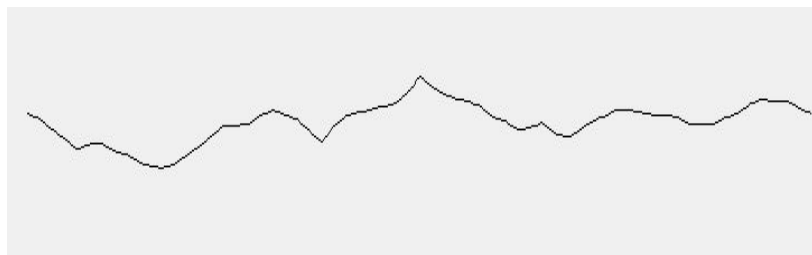
Some of the functions produce output which exceeds the upper scale of the graph, however, the full value is used to modulate the RNG, which sometimes exceeds one, giving heights above the given height parameter. Also, some of the finer details may be lost due to resizing the images.



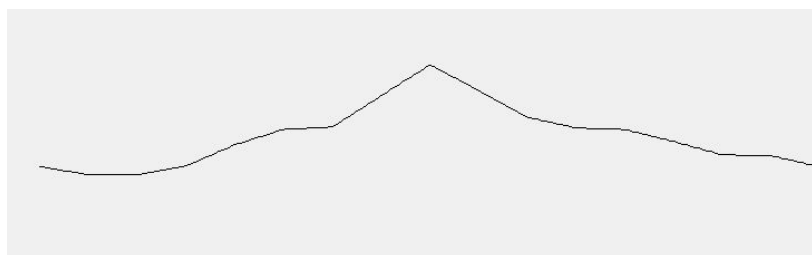
**$\sin(x)$**



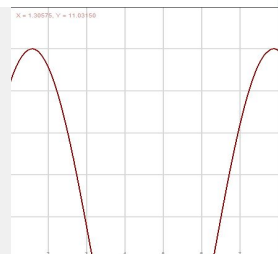
**$\sin(x/2-2)*8$**



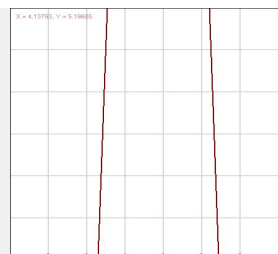
$$\sin(x/2+6)*10$$



$$\sin(x/2+7)*10$$

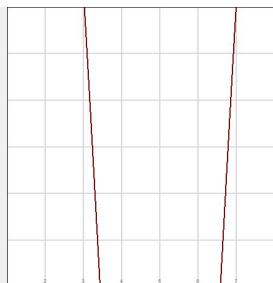


$$\sin(x)*10$$

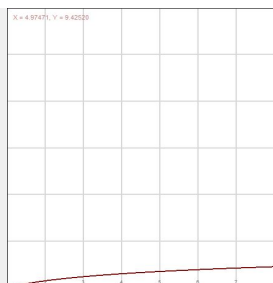
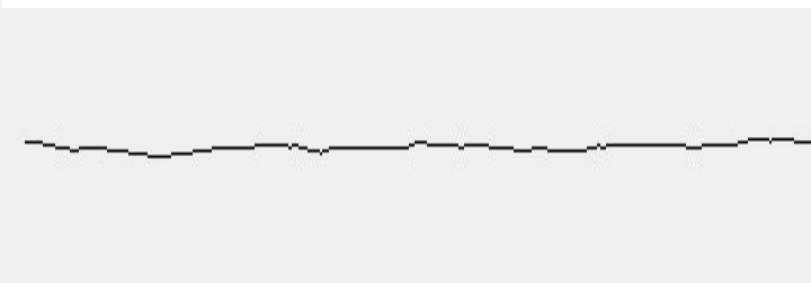


$$\sin(x+3)*50$$

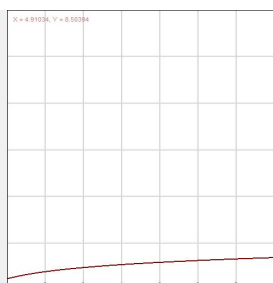
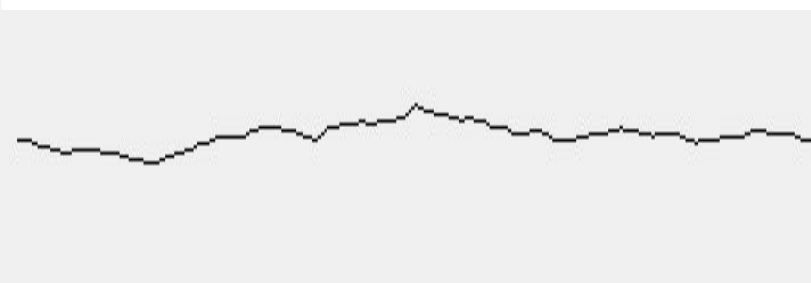




$$\sin(x+6)*30$$



$$\log(x)$$



$$\log(x*3)$$

## Conclusions

This seems to demonstrate that varying parameters can have a significant effect on the output and give greater variety to terrains. The greatest effect is made by randomising the midpoint though this process needs more control. Using functions as parameters allows a great deal of control and gives a good variation in output although a lot on number tweaking is required.

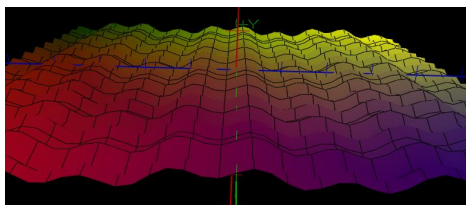
It is now clear how these methods will work with other PCG algorithms such as Perlin noise.

## Further Work

Future experiments will look at bringing more control to the randomized midpoint, perhaps by using functions to modulate the output of RNG. It is also important to adapt these approaches to 3D. The functions used can be applied to 3D as seen in the graph below although it is less clear how that can be applied to a 3D terrain. Possibly the fractal terrain could be summed with the function, or each x z point of the function could be applied to a fractal parameter of the terrain at that point.

It would also be useful to be able to smoothly transition between one function and another over the width, depth and height dimension. This might be possible by having multiple terms and varying the constants used, when a term is multiplied by a constant of zero it is effectively removed from the equation. The constants themselves could be varied by a fractal algorithm.

It would also be useful to try to adapt these methods to other PCG algorithms.



*By Niall Moody*

[illegible]



THAT CUPBOARD LEAPT IF THE ARCHAIC PEOPLE FLEE... IT CONSCRIPTS  
 TEMPTED WITH NAMELESS BE KIND. WE LEARN TO  
 EYES. GIVE GIFTS TO YOUR SIBLINGS. SPEAK MY ART INTO WORDS  
 STEWART, POSSESSED WITH WE DENOUNCE COMMUNICATION. CITY. THAT DESTROY  
 GREAT NUMBERS EVERY WATCH THE FIELD. OUR ART. STEP  
 WAY OF BEING STRUCK WHISPERED SINGING PLAYED THE 2ND  
 THE CIRCUMFERENCE; ALONG LIKE THE GODS OF OLD. EVERY THICK  
 CAN NOT WAS SO. KOOSA, OTTAWA, GESTURE BECOME  
 WE REALLY OUR ACTIONS DON'T WISH MONON & AHELA, DUSTY EYE. I WOULD NOT  
 LOVE CODE THE BE EXPLAINED. TALK TO SAUK, NATCHEZ, SINGING OAKS  
 WITH WAKING. AFTER THE YOUR CHATTAHOOCHEE. TREMBLED BY THE  
 GLASS. PROFANE SONGS WE ARE LIKE A CHOIR. YOU DON'T NEED TO  
 ARE RETURN... CARRY ALL THAT WEIGHT.  
 DESERVING. BE KIND TO THE FIELDS.  
 EVERY ONE OF US IS DEFINED BY OUR RELIANCE ON LANGUAGE. STONING TO LEMEL ALL AROUND  
 FINALLY, DEAREST. IT WON'T BE LONG. WE REGULARLY WOKE IN THE ARMS OF THE MOON.  
 MY HEART BEAT SHADOWED AND AWARE. THERE WAS NO NEAT WONDER, WITHIN HAPPY TIME.  
 STEVENSON'S BIRD OF SUMMER: AS THE MONSTROUS WALLS RISE...  
 SINGS OF BUSY WOODLAND & SILVER WINDS. I TOOK TO THE ROAD  
 WE HAVE SHELTERED IN THE WRITTEN WORD. WE CARESS GLITTERING THE BEES. THE ROAD  
 FLASHING FLEE WILL OUR CONCEPTION OF OUR ART IS SILENCED BY A HUNGRY WITH TIRED HOPE. THE WORDS DAWN.  
 ITCH & HUM AMONG LANGUAGE THAT CAN ONLY ARTICULATE A SMALL FRACTION SINGING STARS  
 CRIMSON RUPTURES. OF HUMAN EXPERIENCE. IN EVERY HEART. WE ARGUE OVER  
 BE CAREFUL AROUND IT WON'T BE LONG. CHILD. THE LEAVES QUIET, OILY MACHINES.  
 THE SHUFFLING SHELL ARE TURNING. HOW ARE YOU? TORCHES SHINE AND DROWN'D.  
 LEST IT HARRY YOU. WE PULSE BRISK WHISPERINGS DEEP  
 THERE WILL BE NO CREASSED WORRY IN OUR HAPPY STORIES. IN THE FOREST. THEIR BELIEFS ROSE  
 WE SING HIDDEN ANNE'S BIRD OF BY THE PLANKS CURIOUSITY: SONG A SHIMMERING, SHUDDERING MAKE GAMES FOR YOUR CHILDREN WOKEN OR  
 AND BOATS. STACCATO. DON'T TRY TO HARD. GUIDED AT ROADS.

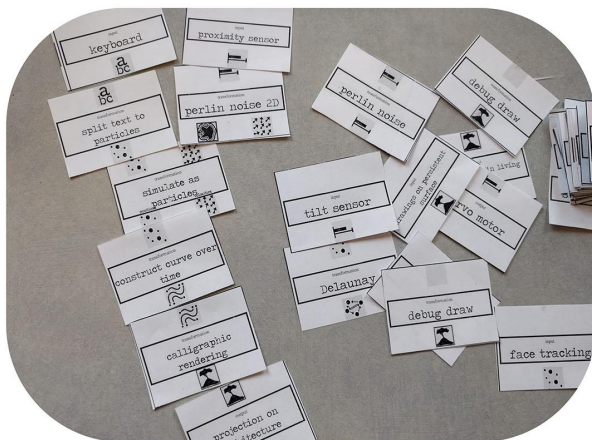
I made the zine by  
 hand-writing/drawing  
 hundreds of  
 sentences generated  
 by the game's text  
 generators (which are  
 a combination of  
 custom grammars,  
 and rudimentary  
 markov chain  
 generators whose  
 inputs are Wasily  
 Kandinsky's  
 Concerning the  
 Spiritual in Art, Walt  
 Whitman's Leaves of  
 Grass, and Robert  
 Kirk's A Secret  
 Commonwealth).



# Kate's CUT N PLAY GENERATIVE FUN CORNER!

Hey kids, want to make an interactive generative installation, but don't know where to start?

Cut'n'use these cards to diagram how a piece of art takes some input, turns it into data, transforms it, and then renders it to the world. Match up the data symbols to build a piece of art that could *\*really\** work!



For practice, try making famous interactive art (like TextRain, or the Treachery of Sanctuary, or IBM's Marchesa dress)

Or try building your own artistic vision, like an ant-controlled markov chain that's projected on a building



