

SEEDS

The background is a complex, abstract collage composed of numerous irregular, overlapping geometric shapes. These shapes are filled with various patterns and colors: some have diagonal blue and white stripes, others feature a green grid pattern, and many are solid colors in shades of orange, yellow, and red. The overall effect is a vibrant, textured mosaic.

Issue 3



Editors: Jupiter Hadley - @Jupiter_Hadley
Dann Sullivan - @FBFDann

Contributors:

Enrica Prazzoli
Frank Mitchell
Yevhen Loza
Nathan Partlan
Magy Seif El-Nasr
Samim
Issac Karth
Genetic Moo
Adam Riddle
Allison Perrone
Mark Rickerby
Antonios Liapis
Spencer Egart
Davide Ciacco

Paul McCann
Seth Alter
Guillaume Pelletier-Auger
Pol Clarissou
Max Kreminski
Jasmine Otto
Damien Crawford
Guerric Haché
Scott Turner
Amin Babadi
Christiaan Moleman
Jo Mazeika
Terry Trowbridge
Joseph Alexander Brown
Denis Kozlov

Matt Schell
Keith Evans
Prophet Goddess
Sabine Wieluch
Juliette Foucaut
Doug Binks
Isaac Schankler
Elle Sullivan
Tilman Schmidt
Martin O'Leary
Johan Rende
Alexander Pech
Gorm Lai
Dan Cox

Organisers:

Joseph Brown, Mike Cook,
Jupiter Hadley, Rachel Hwang,
Azalea Raad, Dann Sullivan

Thanks to: Freehold Games, Janelle Shane,
Matthew Plummer-Fernandez, James Ryan,
prophet goddess, Mark Johnson, Simon
Colton, Adam Smith

Speakers: Berrak Nil Boya,
Bruno Dias, prophet goddess,
Jason Grinblat, Matthew
Plumer-Fernandez, Christoph
Salge, Janelle Shane, Anne
Sullivan

Cover Art by: Martin O'Leary

Some header/footer patterns from Martin
O'Leary's daily drawings. You can find
and buy prints of them here:

<https://twitter.com/mewo2sketches>



Special Thanks

To Our Kickstarter Backers

Without the following people who supported our Kickstarter, we would not have been able to fund the amount of assets, tutorials, talks, and other aspects of the ProcJam. Thank you ever so much from the bottom of our hearts.

Abe Gellis	Caleb Jones	Ed
Adam Hoyle	Carey	Ed Powley
Adam Norton	Carolyn Goodell	Edward Laverick
Adam Smith	Case Portman	Emily Short
Adam Wood	Charles Tangora	Emma
Adrian Gonzalez	CheshireSwift	Emma Lord
Adrien Brizard	Chip	enrica p
Alex	Chris	Eoin Carroll
Alex Clay	Chris Allen	Eric Schmidt
Alex Cook	Chris Janes	Eric Schwarzkopf
Alisblack	Ciro Durán	Erik 256
Andrea Castegnaro	Clayton Cooper	Evan Cobb
Andrew Armstrong	Colin Mitchell	Fed Kasatkin
Andrew Fray	craigp	Feufochmar
Andrew Hollenbach	Cyrill Etter	Forrest Oliphant
Andrew Kalek	Cyrille Morin	Frank Lyder Bredland
Andrew Lim	D.Rail	gabb
Andrew Wang	Dan Tasse	GapGen
Andy Robert	Dana Chayes	Garrow
Anne Sullivan	Daniel Gonçalves	Gary Steinke
Ashley Elsdon	Daniel Nye Griffiths	Gavin Inglis
Ben Howarth	Daniel Tintle	Genetic Moo
Ben Lambell	Daniel Walker	George Rowe
Ben McGraw (Grue)	Delibean	Gorm Lai
Bill Nega	Derek Stobbe	Guerric Haché
Björn Prömpeler	Didier Despois	Guilherme Tæws
BrettW	Doctor Popular	Guillaume
Brian Sowers	Dylan McDiarmid	Gunnar Hemmann
bunnyhero	Dylan Sinnott	harryhatman
Caden Potter	Eclogiter	Heather Corcoran



Henry Peteet	Kemp	Michelle Yeargin
Henry Smith	Kennet Hernandez	Miguel Sicart
Hilary Mason	Kevin Jacobson	Mike Judge
Hodge	Kitfox Games	Mike Powers
Hugh Berglind	Krystal Śmierć	Min Zhao
Hyacinth Nil	Le	Nathan Fritz
Ian Badcoe	Lee Tusman	Nathaniel Mitchell
Isaac Fulkerson	Lemming	netsabes
Isaac Karth	Leo Browning	NICOLAS Jean
István Köhalmi	Liza Daly	Nikki
Jacob Garbe	Llaura	Noah Emmet
James Bell	Lorenzo Grompone	Oddvar Lovaas
James Dreiss	Luke	Oleg Dolya
James Ramirez	Luke Dicken	Pachyderm
James Webber	Luke Weber	Pat Ashe
Jamie Woodcock	Lynn Cherny	Paul Hart
Janelle Shane	maetl	Phoenix Perry
Jason Grinblat	Mardoch	Quinn Monk
Jason Peacock	margaretmoser	r618
Jasper Stocker	Mark Clerkin	Rainer Volz
Jimmy Schubert	Mark Ffrench	Random Commoner
Jo	Mark Gritter	Richie Camara
Joerg Reisig	Mark Renner	Rick
John Hergenroeder	Mark Wunsch	Rob Haines
Johnathan Pagnutti	Martin Kugler	Rob Saunders
Johnicholas Hines	Matt Craven	Robert Masella
João M. Cunha	Matthew	Robert Wells
JReynolds	Matthew Deline	Robin Baumgarten
Julien Delezenne	Max Silbiger	Roc
Julio Terra	megallo	Roman Panasko
Jupiter Hadley	Michael Gradin	Romeo Barnett
Jurie Horneman	Michael Langford	Rory
Justin Loudermilk	Michael Springer	ruby
Kate Compton	Michael Twomey	Russell Fincher
Kayn	Michelle	Ryan Orlando





Sam Humleker
samtb
santi.ontanon
Scott Anderson
Scott Grant
Scott Lininger
SeaWyrn
Several
Shaddock Heath
Shane Celis

Shawn Taylor
Stephane
Steve Mumford
Steve Washington
Sven Nilsson
T. A.
Tam Toucan
Terry Smith
Thomas Smith
Thomas Winters

Tieg Zaharia
Tim Plummer
Tom
Tommy Thompson
Travis Faas
Troy Visineau
Tyler Coleman
Will Stavely
Wolf Owczarek
ZAE

Thank you to everyone who shared our Kickstarter and who brought light to it! For ProcJam 2019, we will also have a Kickstarter, so keep an eye out if you want to support us!

PROCJAM

Make Something That Makes Something



ProcJam

Make Something That Makes Something

The ProcJam is a unique, relaxed game jam that aims to make procedural generation accessible and to show off projects that are pushing the boundaries of generative software. As a whole, this jam is laidback, easy to enter, and fun to be apart of. We are working to build a community of friends and peers across disciplines all interested in procedural generation.

The ProcJam takes place across nine days, including two weekends. You can enter anything you'd like - art, video games, board games, tools, anything you'd like to create as long as it has something to do with procedural generation/random generation/generative software etc. You could even take an existing project and add some generative magic to it for the jam! If you start before the start of the jam or enter your project after the end of the jam, that's fine as well.

ProcJam is also more than a game jam - with the help of our Kickstarter Backers, we are able to fund art packs, tutorials, and talks all on generation. These resources are put out publically as ways to grow the community and help get people into generation.

This is truly a community effort, even down to this zine which was made from submissions from the ProcJam community.

We hope you enjoy it!



Table of Contents

Yet Another Contemporary Jewellery Exhibition.....	3
Writing Interesting AI.....	6
What's Your NAME?	9
Towards Generating AI for Generative Worlds: Evolving Behavior Trees in Unreal Engine 4.....	12
Thoughts on the Generative, Creative Economy.....	16
The Locus of a Generator.....	19
Robots on Typewriters.....	22
Deciphering Generated Symbols as a Game Mechanic: The Design of "Alien Transmission".....	25
Bugs & Beetles.....	29
On the Responsibility of Makers.....	32
A Web-based Node Image Editor.....	34
Read-world Data as a Seed.....	35
Representing Writing.....	41
Making a Fantasy Newsroom Bot.....	45
Procedural Generation in Twine.....	47
A Year of Daily Generative Sketches.....	49
Procedurally Generated Spellbook Sprites.....	52
Procedural Generation, Variety and Reproducibility.....	53
Letting Go of Realism.....	55
Mountscapes.....	58
Mere Juxtaposition - On Generative Fiction.....	61
Intelligent Middle-Level Game Control.....	69
Superorganism Art.....	70

Not_constantinople.....	74
Nos Falaises.....	75
Minus World: Generative Game	
Reviews from a Parallel Universe.....	77
Oddifier: RPG Character Sheets.....	80
All Dinosaurs are Great and Small.....	82
Story Generation by Algorithms: A	
First Attempt at a Digital Storyteller.....	86
Islands Are Just Mountains Up To	
Their Necks in Ocean: Part 1.....	90
Guppy - Procgen as Antidote to Development Boredom.....	96
Game Randomizers: Procedural Variations on Your Nostalgia.....	99
Designing Strata: A 2D Level Generator for Non-Programmers.....	101
Experiments in Generative Geometry:	
Building Meshes in Unity, Vertex by Vertex.....	105
From Hacker Poets to Cybernetic Poets.....	108
Boxes in Space.....	110
Carving the Infinite Plane into Discrete Regions.....	117
Islands Are Just Mountains Up To Their Necks in Ocean: Part 2.....	121
Apop: Seeing Patterns Everywhere.....	131
Alien Growth - A 2D Plant Generator.....	133
But What Do you *Do*: Mechanical Ideas for	
Turning a Cool Generator Into a Compelling	
Game.....	135
High End Procedural Systems (Procedural Trip Report).....	138

Yet Another Contemporary Jewellery Exhibition

By Enrica Prazzoli

www.enricaprazzoli.com | @enricapr

My name is Enrica, and I am a contemporary jewellery artist.

Today you are going to visit a contemporary jewellery exhibition.

“Contemporary” as in contemporary art; “jewellery” in the widest and wildest sense, as in “things that go on the body” and/or are body related. You might have never heard of it before, and that's fine: it's a pretty specific field.

You enter the place where the exhibition is hosted, and find yourself in a <<trace "place">>; you look around, and you see <<trace "visitor">>.

You might wonder, how I got from making things that go on the body, to making generative text?

“Stop thinking about art works as objects, and start thinking about them as triggers for experiences.” Roy Ascott

Sometimes the starting point of a project is the kind of experience I want to create; sometimes it is something else, but thinking about the experience is something that always happens during the process.

You get close to the jewellery display and <<trace "adv">> <<trace "verb">> a piece that is <<trace "size">>. You get closer, to a have a look at this <<trace "Jewel">>.

In this case, I wanted to create an experience that was playful, familiar to the people who have an interest in the field, is always different, but also feels like it's always the same. By using text only, in a boring format (Verdana, black text on a white background, on a laptop, not a custom-made installation) I leave space for the reader to project their own personal version of whatever random combination was generated especially for them.

“Sometimes the starting point of a project is the kind of experience I want to create; sometimes it is something else...”

You move on to the next piece; this one is <<trace "size">>. You <<trace "adv">> <<trace "verb">> <<trace "Jewel.a">>.

I've seen it happens in different fields, and I've experienced it myself several times : you discover something that's new to you, get interested in it, you look out for more, marvelling at the existing variety of methods, techniques, outputs, things that can be done within that field. You keep looking for more, keeping up with what's happening, looking back at the history of that subject. And after a while (maybe months, maybe years, maybe decades) you start feeling that, yes, endless variations exist, but it mostly feels like more of the same old thing and it's really rare that you are truly surprised.


The next one is <<trace "size">>. You <<trace "adv">> <<trace "verb">> it: it appears to be <<trace "Jewel.a">>.

“Yet another contemporary jewellery exhibition” is a text-based interactive installation that takes the form of a procedurally-generated text on a webpage, describing a visit to a hypothetical exhibition. The only interaction possible for the visitor is to refresh the page, creating a new version and forever erasing the previous one.

You move forward to look at one more piece, <<trace "size.a">> <<trace "Jewel">>.

Narrated from a second-person point of view, it briefly describes the experience of a visitor entering the gallery space, looking at the exhibited jewellery pieces and, finally, briefly examining the feelings towards contemporary jewellery elicited by this visit.

You reach the last piece of this small exhibition: you <<trace "adv">> <<trace "verb">> <<trace "size.a">> <<trace "Jewel">>.



I have used Twine 1.4.2 because I like Jonah as a format, and at first I thought that showing one paragraph at a time would create an experience closer to the one of being in a gallery and looking at one artwork after the other. However “Yet another contemporary jewellery exhibition” was going to be actually shown in a gallery, and the gallerist agreed with me that, in this case, that was redundant. So you get all the text at once. I used Twinecery for the generative part, and I found it way more forgiving than straight up Tracery.

You exit the place and stop briefly to consider what you have just seen.

It is a way to playfully explore my sometimes conflicting feelings towards this field: the enthusiasm and the energy and the ingenuity that often permeate self-organized shows; the feeling of information overload after visiting several exhibitions in a row during jewellery weeks; the excitement of working on a new project; the feeling of dread that everything has already been done, better than I will ever be able to.

After this visit, you feel that you <<trace "lovehate">> contemporary jewellery.

Writing Interesting AI

By Frank Mitchell

<https://www.frankmitchell.org> | @onefrankguy

For a long time, the only kind of game AI I knew how to write was a mean one. You know, the kind that picks the best possible move every time. The kind no one wants to play against.

It wasn't until I found Dan Cook's work on game atoms, and player skill progression in games, that I started thinking there might be a way to write less mean AI [1]. Maybe computers could learn to play games more like people, and I could write AI that were interesting game players.

Any rule book for a game has two things at its core:

1. The mechanics of how to play
2. The win and loss conditions

There is almost never anything about skill, or strategy, or tactics, or getting better. We learn these things as we play the game. So how do you build an AI that can learn those things too?

You start at the same place you do when teaching a person. These are the things you can do on your turn. This is how you know when the game is over. Code the game state as raw data. Code each "thing you can do on your turn" as a function that takes an existing game state and transforms it into a new game state. Code "how you know when the game is over" as a function that takes a game state and returns yes or no. Now you have enough for an AI.

In the beginning, your AI will play randomly. It will pick a thing to do on its turn and it will do that thing. Then it will

check to see if the game is over. Think about a first time checkers player. They push pieces around the board. They follow the rules. Take a jump if you can. If there are no jumps to take, move diagonally to an empty space. This is what your AI does too. It is a beginner with no memory. It doesn't see forks or double jumps or king making moves. How do you teach it those things?


With just a rule book, the most basic assessment of a game you can make is, does this move cause me to win? So have your AI do that. Get a list of all the possible moves. If any of them are winning moves, pick a winning move to play. If none of them are winning moves, pick a random move to play.

If you can see moves that cause you to win, you can also see moves that cause your opponent to win. Have your AI do that too. Find all the winning moves that the opponent could make if it was their turn. Find all the moves the AI can make that will block the opponent from winning. Have your AI choose a random blocking move and play it.

Now you have an interesting AI. If it has a winning move, it plays one. If it has no winning moves, it plays a blocking move. If it has no blocking moves, it plays a random move. But it's not perfect. A good player can still beat it. When you set up a fork, where you have two winning moves, the AI can only block one of them.

Each of these decisions for move types (winning, blocking, random) is a function. Think of more functions to make your AI more interesting. How about a function to pick moves that result in double jumps? Or a function to pick moves that

"If it has a winning move, it plays one. If it has no winning moves, it plays a blocking move."



aren't back row pieces. Or a function to pick moves that make kings. Now your AI can do lots of things, and the order it chooses to do them in becomes its style of play.

Your AI might be flashy, choosing double jump moves and king making moves. Or it might be defensive, choosing blocking moves and moves that keep back row pieces immobile. Or it might be chaotic, trying every move function randomly until it finds something playable.

In the end, your AI might ignore winning moves entirely. So when it wins, it will be by accident, almost as if it didn't see the winning move. Almost as if it made a mistake. Almost as if it was human.

P.S. If you're looking for code examples, I made an interesting AI for Nine Holes, one of the oldest games in the world [2].

-
- [1] <http://www.lostgarden.com/2007/07/chemistry-of-game-design.html>
[2] <https://onefrankguy.github.io/nine-holes/>

What's Your NAME?

By Yevhen Loza (EugeneLoza)

In roguelike or sandbox games procedural generation is used not only to create world areas and fill them with items and enemies. Friendly NPCs also want to be a part of the game and to do so, they need a worthy name. In this article I'd like to share a simple names and nouns generation algorithm I've made this summer.

A name is not a random set of letters. It's a word. Maybe, it had lost its meaning long ago, maybe it came from a long forgotten folk. But most often it follows the normal word building rules in native language. Therefore it looks like a good idea to build names from a usual dictionary.

"A name is not a random set of letters. It's a word."

Word Structure

First of all, let's make an important simplification, so that we won't be forced to analyze each syllable in a grammatically correct way. We may imagine each word structure as "(C)VC+VC+VC(V)", i.e. each "syllable" being a vowel+consonant block (VC). The "first syllable" may also have "leading consonant block", and therefore should be stored and used separately. Let's also demand, that there should be no syllables with zero consonants to avoid generating names like "Miisaaaal".

Then we split each words into "syllables" and store them in an array or a list. However, some syllables are used frequently, and others are rare. Therefore the algorithm should also store the number of "hits" per syllable, so that generated words would follow the same pattern.

Namespace

You know, I have a very common name for a place I live in. And it took me quite some time to stop reacting to random people around

calling my name :) So, we should avoid that problem by creating a "namespace" we're all used to in programming - no two variables may have equal names. So, while generating the name we just check it against the names that have already been generated.

Moreover, we've just used almost a million words to build our set of syllables and it's quite possible that the generated word will be... yes, exactly one of the words that was used to generate it. So, right after we've created the list of words, we just add them to namespace to avoid NPCs called "Rhinoceros", "Skyscraper" and "Watermelon".

Ban list

I know a Chinese scientist whose name sounds like [CENSORED] in Ukrainian language. Such things happen in real life and often result in stupid jokes and puns.

We should try to avoid such inconveniences by creating a "ban list". It works similar to namespace, but while the namespace checks the whole word, the ban list scans all the substrings. We don't want a character named "Mol[CENSORED]as" in our game, don't we? The probability is not as low as one might think and concerns about 2% of all possible names.

But of course, only human ear can tell what sounds nice or ugly. So, we may also create a manual ban list of letter combinations like "rdsp" or "thyth" which are hard to pronounce, but may by chance appear in generated names.

We should also control consonant-to-vowels ratio in words, so that there would be not too many complex consonant blocks. Average 2-4 letters per vowel in a word produces quiet nice results.

"It works similar to namespace, but while the namespace checks the whole word, the ban list scans all the substrings."

The end?

The last thing remaining is to add the word ending. With "VC" syllables structure we guarantee that the word always ends with a consonant. Such generated name suits a male character and adding a letter "a" in the end makes a female name. We may also use a simple trick to make short names (nicknames) for our characters by just using their first syllable and adding "i" letter in the end (e.g. "Lepuha aka Lepi").

Result

In an example below 4 Public Domain licensed English word lists were used with total of 900660 words, also ban lists were created. Processing them with the algorithms described above produces 11760 first syllables and 1802 normal syllables. On average desktop PC each name takes about 16-18 ms to generate.

"With up to 4 syllables per word that provides for over a trillion of unique names."

With up to 4 syllables per word that provides for over a trillion of unique names. And of course, this algorithm may be used not only for names in English, but for most other languages too. And let's welcome our new NPCs named Inscrana, Bernof, Phimowax, Baltela, Sniggit, Kagudum and Degluca!

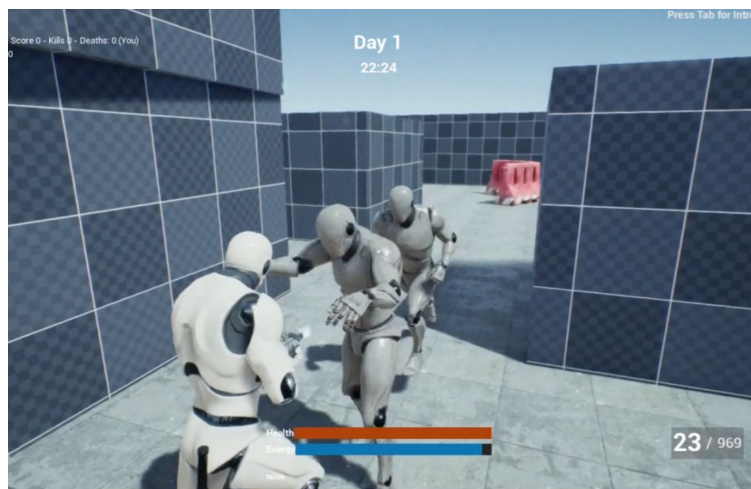
You may find the corresponding FreePascal code and word lists here: <https://gitlab.com/EugeneLoza/DecoNouns>

Towards Generating AI for Generative Worlds: Evolving Behavior Trees in Unreal Engine 4

By Nathan Partlan and Magy Seif El-Nasr

npc.codes | @NPCDev

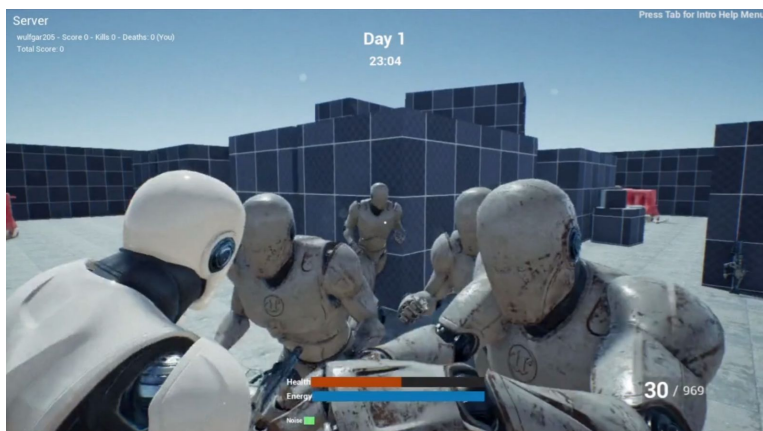
Many games procedurally generate significant portions of their environments, gameplay, and even narrative. Generated game environments are not new; roguelikes and other games using procedural content generation (PCG) have existed since almost the inception of the medium. Recently, however, high-profile titles such as *No Man's Sky*, *Fortnite*, and *Shadow of War* have relied heavily on PCG.



With this focus on generativity, there is a need for improvement and research in designing and building compelling, capable AI agents to populate their complex, generated worlds. We are experimenting with one approach to help designers build more comprehensive and robust AI that may be more capable of tackling decision-making in generated environments.

The AI for *Fortnite* was one of the subjects of a GDC talk (Isla and Abercrombie 2016) about the challenges and possible approaches for AI in a generated environment, focusing on movement and pathfinding. These are important problems, and tractable ones for

which the Fortnite team found solid solutions. However, the agents in question were limited to movement and combat behaviors. For generative games to become more inclusive and expand into new genres, we need to develop affordances beyond those, building more compelling narratives and enabling creative gameplay and problem-solving for players.



“...we built a genetic programming system for Unreal Engine 4’s behavior trees.”

To begin to approach these problems, we are building on research on evolving behavior trees (Perez et. al. 2011). Our approach applies genetic programming to modify hand-designed behavior trees, a commonly used architecture for controlling game AI agents. By starting from designer-specified behavior trees, and then generating evolutionarily advantageous improvements to them for unforeseen situations, we hope to retain designers’ goals for their agents while building more robust behavior for generated environments.

Genetic programming is a technique for evolving trees, usually applied to “parse trees” for computer programs (Poli et. al. 2008). It naturally maps to behavior trees, which operate much like simple programs. Therefore, we built a genetic programming system for Unreal Engine 4’s behavior trees. The details are outside the scope of this article. However, the process involves mapping the behavior

trees into a “library” of possible node types, simple crossover and mutation operators, which swap nodes between trees and randomly replace nodes with other known node types, and a fitness function to evaluate the resulting trees. We use a sped-up version of real gameplay for evaluation.



To test this system, we generated behavior for “zombie” AI agents in the open-source “*Epic Survival Game*” by Tom Looman. We removed functionality for chasing and attacking from the zombies’ behavior trees, but provided a library of useful nodes to our evolutionary algorithm. We found that, within a few generations, zombies would learn to chase the player and patrol the area. At first, some zombies would attack the player briefly, then turn and run away. By adding the size of the tree as a fitness penalty, we found that the efficiency and simplicity of the trees would improve over time, until most zombies exhibited consistent chasing behavior. Although this was a very simple test environment, we believe that future iteration will allow us to generate behavior for more varied and complex agents. Our next step will be to separate our system into a generic library for behavior tree evolution in Unreal Engine 4 so that it can be tested with more games.

***“We will need
to provide
visualizations
and
user-friendly
tools for
designers to
understand...”***

In the future, we will need to perform further research to make our system easier to use for designers. We will need to provide visualizations and user-friendly tools for designers to understand, select between, and debug the resulting behavior trees. If designers feel that their control is infringed too heavily by generative tools, they are unlikely to trust and use them. Moreover, defining effective fitness functions for complex agent behavior is a difficult and under-studied area of game AI research. We will need to develop new methods for defining and tweaking fitness functions and controls for gameplay evaluation in order to make this tool accessible. Finally, we need to be able to test generated agents in realistic gameplay environments, which means developing models of player behavior with which to train the agents. Only by combining effective player models, generative worlds, and designer-friendly tools can we enable responsive AI that works with humans to create a more inclusive, expansive future for games.

Acknowledgements

Thanks to Isha Srivastava and Alex Grundwerg, who contributed code and ideas to this project.

References

Isla, Damian, and John Abercrombie. 2016. “AI For Generated Worlds.” presented at the Game Developers Conference. <https://gdcvault.com/play/1023418/AI-For-Generated>.

Perez, Diego, Miguel Nicolau, Michael O'Neill, and Anthony Brabazon. 2011. “Evolving Behaviour Trees for the Mario AI Competition Using Grammatical Evolution.” In Applications of Evolutionary Computation. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-20525-5_13.

Poli, Riccardo, William B. Langdon, and Nicholas Freitag McPhee. 2008. A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd.

Thoughts on the Generative, Creative Economy

By Samim

@samim

I find it comedic how professional digital creative tools (video, music, design, etc.) in 2018 are still being mainly promoted as "tools for hand-crafted, artisanal creation" - while we are living in an age where bot-nets are producing and recommending artefacts by the trillions.

While the quality of creative artefacts generated with machine assistance is debatable — it is quickly getting "good enough" across many fields. Combined with other unique benefits of the generative model, traditional digital creative tools are becoming obsolete fast.

The outlines of the new creative economy are clear: 90% of all content will be machine generated (with little to no human intervention) and extremely cheap to buy. The remaining 10% will be artisanally made by humans, which focus mainly on branding and storytelling. Technologically, you don't need "AI" or very sophisticated machine learning for most of this to happen — good old creative computation (at massive scale) does the trick.

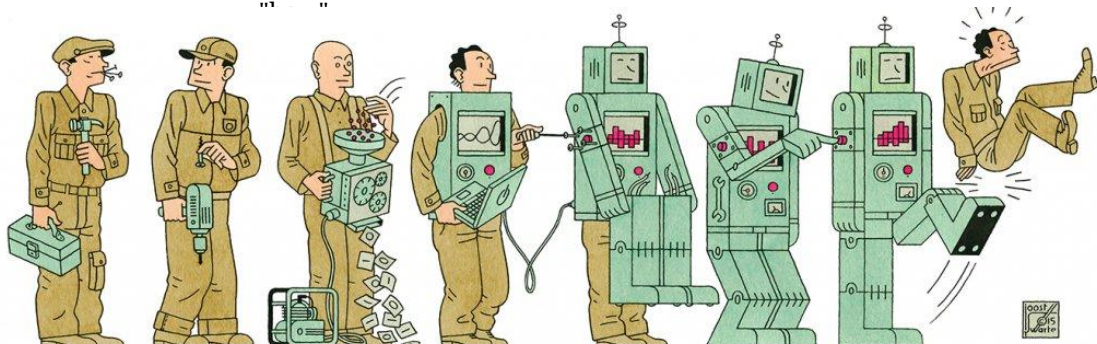
Finally, a note to the people saying "No worries, AI + creativity is all about empowering and augmenting human creators, there won't be blood." I've worked in this industry for years & find such talk intellectually lazy and at times even deceptive. There *will* be blood.

I am not questioning that AI can and is empowering some creators and enabling wonderful new forms of creation — it will. But, to see what happens with such tech when at industrial scale, one can look at fake-news and spam generation bots. That's a more realistic view of what's coming to all creative industries, operating under our current global economic paradigm.

The generative creative economy won't be a utopia by any stretch, but is likely a messy hell-hole for many creatives — not unlike the

situation right now. We have to actively shape this future by fighting vested interests in the system. Yes, human jobs destroyed through growing automation capabilities can be replaced with entirely new jobs. But it won't happen magically — it's a major generational effort which requires deep investment in education and culture. Human creators need to be recognised, not hidden, by AI. Under the current klepto-capitalist global system, that seems highly unlikely.


If we want more positive outcomes, the "why" needs to be addressed properly, beyond our current semi-pornographic obsession with the
" " "



Beyond doom and gloom, there is hope: The vision/opportunity I sense is phenomenal: Creative AI tools could help raise the global level of literacy, creativity and empathy, by dramatically shortening the time of thought to highly communicative (possibly multimodal) artefacts.

Many moons ago I wrote this extensive piece on the sector, from a more hopeful, slightly utopian perspective:
<https://medium.com/@creativeai/creativeai-9d4b2346faf3>

Yet the reality of creative industries today looks very different: Advertising runs the show (see Google etc) — which is really just a synonym for either "spam" or "social engineering": Both horrible for



societies in times of climate change — and not conducive as objective functions for more creativity or education. In the sense of raw reach & economic impact, the advertising industry is arguably the most powerful "art form" of the 21st century. And so when you ask for "concrete example" of dystopian generative systems, we must look at developments in this area.

One of the most influential developer of creative tools — Adobe — has recently doubled down on the intersection of "AI" and "Creativity", with their "Sensei" Initiative. Guess who they are targeting with their tools? Yes, Advertisers. In the sector of advertising, the quality VS quantity debate is very different: it is NOT about the values of art in the classical sense, or about storytelling or other humane things — it is about naked money. This is a clearly parameterized signal to feed to machines as loss/reward-function, and due to this fake clarity, such systems can grow like cancer.

Such economic success is precisely what drives humans to extend generative advertising logic to other creative fields. Repetitive generative muzak in the charts and endlessly boring superhero movie rehashes on the screen are a reality already today, and all using some form of "optimisation" algorithms. And so we see the rise of "creative" botnets - the rise of fake news and ever more fractaline simulacra's (see this post: <https://samim.io/p/2018-04-23-the-solution-to-exploding-fake-news-and-spam-botnets-ca/>)

"Such economic success is precisely what drives humans to extend generative advertising logic to other creative fields."

I personally do not agree with the politics and dynamics of the visions outlined here. My writing is meant as a warning pointer within search space. We must strive towards more appealing outcomes, for humanity's sanity, and well-being, in times of climate change might very well depend on it.

The Locus of a Generator

By Issac Karth

<https://procedural-generation.isaackarth.com>


There's no generator that is the best for every circumstance. Indeed, you often want to tailor the generator to fit your individual goals. One lens that we can use to analyze how generators can differ is what I've taken to calling the generator's Locus. The name is based on the idea of the center of attention or the center of gravity: where is the locus of control for this generator? Where are we focusing our analysis?

For example, in Minecraft the generator places diamonds deep in the earth, following a specific set of rules: they have to be deep but not too deep, they are placed in clusters, each cluster is generated at a distance from other clusters, and so on. Once they're mined, the diamonds are basically all the same. Their difference comes from the structure of the way they are generated. We can call a generator that speaks to the player through the structure of generated things one with a locus centered on structure.

"Their difference comes from the structure of the way they are generated."

Structure-locus generators tend to be experienced indirectly. The pattern of diamond placement in Minecraft isn't something the player can directly interact with: the player can only directly interface with the individual voxels, not the intangible generator that placed them. Another example is the history generation in games like Dwarf Fortress and Caves of Qud. The player sees the results of the generation, but can't directly view the process that created it. Instead, they infer the relationships that the structure exposes.

This can manifest in many interesting ways: because diamonds in Minecraft have the constraint that they are only found deep underground, the player will encounter them late in the progression. Spelunky always puts the exit lower than the entrance. And players can learn more complex associations: the weapon spawning rules in




PUBG and DayZ give players a way to anticipate where to look while still not guaranteeing any particular outcome. (The military weapons tend to spawn in the army base, the shotgun in hunting lodges...) By using different distributions of probabilities, including some predictable elements, and having constraints that let the player anticipate correlations, the generator becomes a richer experience.

But, other generators don't foreground the creation process. Instead, they concentrate on making the most expressive generated artifact. My favorite example for this is the jetpack platformer *Exile*, originally on the BBC Micro in 1988. To fit a big map on the small disk, they built a generator for it. Many games of the era did similar things, including everyone's famous touchstone, *Elite*. What was significant about *Exile* is that it generates exactly one map. There's no variation and therefore no way for the player to figure out what the generator is capable of creating outside of that single map.

While *Exile* is an extreme example of a one-off generator, there are many other generators that primarily focus on the surface of the generated things. In these cases, the generated thing is more important than the system that generates it. This is a relatively popular way to approach designing a generator, where you want your generator to create interestingly unique and complex things. Some games that use it effectively are the way that *Desert Golf* has one fixed sequence of levels that is the same for all players, the way that *Animal Crossing* towns are generated at the start of the game so they're unique to each player, and any generator that creates a handful of highly-detailed artifacts.

This is complicated by the way that generators can be layered. For example, in *Dwarf Fortress* the dwarves can engrave the stone walls and floors with images, which have procedurally-generated descriptions. An elaborate engraving has a very surface-locus description, with details about what the picture shows. But the pictures are often depictions of historical events from the generated



history, tying them into the structure-locus. The same generated artifact can be viewed from two different perspectives.

Tying the different generators together like this lets us build nested layers of generators that are more meaningful than any single generator would be on its own. As with the engravings from Dwarf Fortress, the histories of the sultans in Caves of Qud are used with nested generators. They are tied into the backstory lore and to places and items that the generator adds to the world. Spore is built around this idea: anything you encounter in the world was made with one of the other generators, often as designed by another player. The individual from the creature creator becomes the template for an entire species.

***“The individual
from the
creature
creator
becomes the
template for an
entire species.”***

Hopefully, thinking about the locus of a generator like this can help us design better generators, find new ways to plug different generators into each other, and gives us new vocabulary to use when we’re performing criticism.

Robots on Typewriters

By Justin Edwards and Allison Perrone

@RobotTypewriter | batcamp.org

Hello, we're Justin and Allison.

We're the hosts of Robots on Typewriters, a new podcast all about computer-generated and computer-assisted comedy. On Robots on Typewriters, we love to laugh about the wild, weird, and sometimes all too human things that come out of random generation, artificial intelligence, and automation. And because the world needs more of that stuff, we always try to make some of our own artificial humor as well.

Who would make such a podcast? Let us introduce ourselves.

Justin is a graduate student at University College Dublin where he's researching conversation agent interactions (talking to Alexa and Siri) and interruptions in those interactions. Justin has been laughing at computers for most of his life, starting by delighting in the procedurally generated names of rookies in sports video games throughout his childhood. He started laughing at computers in a more professional setting during his junior year of college when he took his first human-computer interaction class and started researching multitasking behaviors during computer use. Professionally, he hopes to be an industry researcher that laughs at conversation agents for the foreseeable future. His favorite part about making Robots on Typewriters is, inexplicably, formatting datasets and feeding them to neural networks. What a nerd.

Allison is a writer and otherwise general media creator who works at home full-time and therefore craves the company of computers and the internet to stay sane. Though she's a little less academically qualified than Justin when it comes to laughing at computers and maybe doesn't quite get how it all works, she sure gets why it's so funny. As a child who skirted the edges of mid-2000s internet culture, she's come a long way from thinking how hilarious it is to be

***"...a new podcast
all about
computer-generated and
computer-assisted comedy."***

“so random” and yell “Waffles!” in a room of friends. Now that she understands real comedy, she knows it’s much funnier to click through a Random Food Generator on RandomLists.com and contemplate the absurdity of a meal consisting of only parsley and condensed milk.

The idea for this podcast goes back several years to a night when we found ourselves weeping with laughter, having hijacked Justin’s sister’s phone to text her unwitting friends nonsense messages using her predictive text. Ever since, our interest in comedy and art that utilizes things like procedural generation and other bits of AI has grown exponentially. There is something so endearing about a neural network doing its best to produce a list of plausible band names but turning out results that include “Stritty Landy Halking Mobil Radpian” and “Tamont Clirf.” It’s doing its best.

Randomization perpetrated by a computer adds an element of absurdism to comedy that we can’t trust our own overthinking human brains with. The results of a simple random generator are so pure and thoughtless. We think of those generators, those neural nets, those algorithms as our unseen collaborators who are truly the heart of our podcast. Neither of us are trained comedians and we’re not the most gifted improvisers. Why wouldn’t we hand that hard work off to computers like the millenials we are?

Our show consists of two segments (both named using random word generators): the Zesty Hat and the Trashy Toy.

In our Zesty Hat, one of us presents something interesting or hilarious that we recently found around the net, including things like Twitter bots, blog posts about neural networks, and we even did a mini-feature on ProcJam 2018, showcasing some of our favorite submissions! Inevitably, we end up following more robots than people on social media and exposing each other to our new favorite synthetic personalities on the internet.

For the Trashy Toy, one of us devises a game for the other involving all sorts of computer generated content. Our Trashiest of Toys have seen us attempt the following:

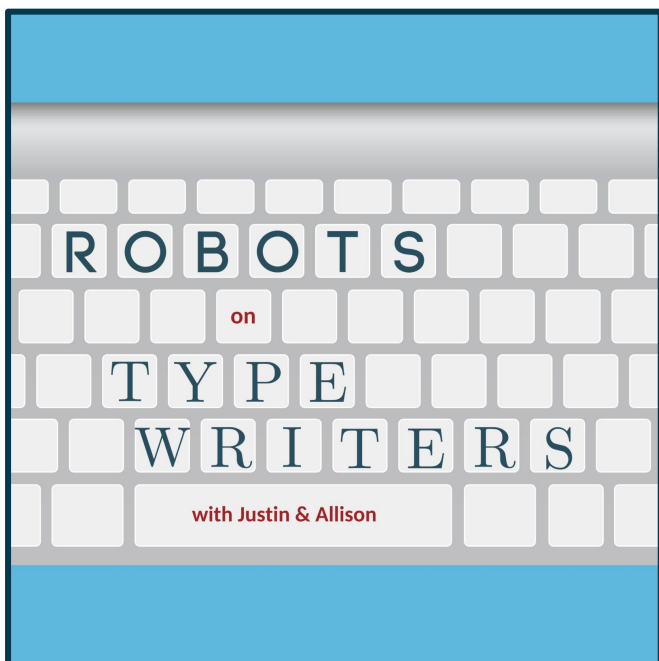
Telling the difference between real college mascots and a neural network's new proposals

Beating IBM's Watson in a high-octane cooking challenge

Sorting real Lil Pump lyrics from a predictive keyboard's version

And if you listen to the very end each week, you'll often hear something a little extra we like to call the Least Significant Bit.

Robots on Typewriters is available on Apple Podcasts, Stitcher, and at batcamp.org. If it piques your interest, please listen, subscribe, and connect with us on social media!



Deciphering Generated Symbols as a Game Mechanic: The Design of "Alien Transmission"

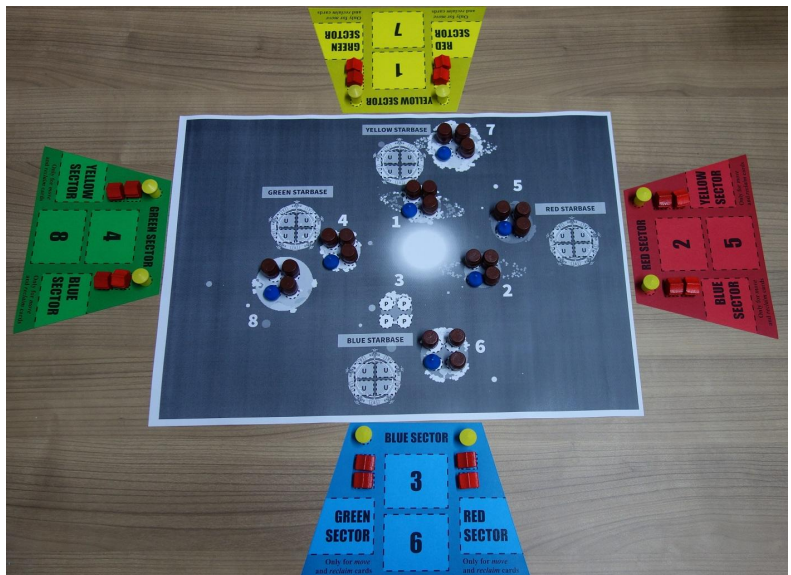
By Antonios Liapis

<http://antoniosliapis.com> | @SentientDesigns

Board game play is enhanced through technology for the purposes of time-keeping in games such as *One-Night Werewolf*, logistics for e.g. hit point tracking in *Hero Realms* or revealing hidden information in *Alchemists*, new mini-games such as the digital puzzles in *Mansions of Madness*, and randomization.

Randomization could be considered a form of procedural content generation, although it often follows a simple digital variant of shuffling a deck or looking up values in a table. While randomization speeds up preparation in long games such as *Mansions of Madness* and ensures information stays hidden in *Alchemists*, it does not particularly contribute to novel gameplay. The generated content (i.e. the event sequence or board) forms a backdrop for traditional game mechanics (dice-rolling, token placement, etc.) and only indirectly affects the player experience.

"While randomization speeds up preparation in long games such as *Mansions of Madness* and ensures information stays hidden in *Alchemists*, it does not particularly contribute to novel gameplay."




In order to explore different uses of generated content in board games, I designed a game ultimately titled *Alien Transmission* where symbols are generated and players must use them to communicate information to each other. This 4-player game has a science-fiction theme and hinges on co-operation for survival against an external invasion. Each player controls one sector of a solar system with two planets (one outer, one inner) and one starbase. Planets can hold population, fortifications or infiltrated aliens, while starbases can hold spaceships. Planets with no population are considered fallen, and if all players' planets fall the game is over. In the current prototype, players must choose one action per turn (from a set of 12 actions) and choose one of their two planets or neighboring sectors where this action is issued. Actions include building spaceships to send to the starbase, sacrificing population to guard against alien threats, sending population or spaceships to players of neighboring sectors, etc.

Actions are taken in response to upcoming alien threats, which may add infiltration tokens to planets (replacing population, if the planet's capacity is full), remove spaceships or population under circumstances (e.g. if there is no fortification or if there is an infiltrated alien on a planet), halve a planet's population unless a quarantine player action is taken, etc. Each player places one action card face down on the relevant planet or neighboring sector on their control panel. Players reveal their action cards simultaneously during the resolution phase, after the alien threat has been shown. Players win if a deck of 20 alien threats is depleted and at least one planet has not yet fallen.

While the board, player actions and randomized alien threat deck are reminiscent of many co-operative survival games such as *Pandemic* and *Arkham Horror*, the key mechanics of *Alien Transmission* revolve around one player's advance knowledge






of the threat and their efforts to warn some or all players. In every turn, one player reads the next alien threat card and must choose which player(s) and which message to transmit to them in order to take the right actions to counter it (if possible). The message itself is where procedural content generation comes in: messages consist of one or more symbols with no meaning ascribed beforehand. The symbols themselves are in black and white and generated via cellular automata with the occasional hard-coded symmetry, rotation, reflection etc. [The codebase for generating your own symbols, and other possible uses, is found at <https://github.com/sentientdesigns/aliensymbols>.]

A pool of 12 different symbols (with duplicates) is available to all players in each game: players may need to develop a common code to decipher them through prolonged interaction during the same game. Players are not allowed to speak — at least not to discuss the symbols themselves or talk about the alien threat. As an example, the first player may wish to notify another player about an upcoming alien virus threat to that player's inner planet: they choose a vaguely skull-shaped symbol and a teardrop-shaped symbol for a message transmitted only to that player but visible to all. Even if the other player fails to decipher it and counter the threat (e.g. via a quarantine action), during resolution all players see the alien threat card and figure out what the message was about. Optimally, other players should use the same symbols in their turns if they wish to communicate a virus attack (although message details such as the planet it is aimed at may change).

The role of procedural content generation in this game is not to randomize the initial state or to handle game-state progression (both are handled in a traditional way by shuffling the alien threat deck). Finding patterns in the randomly generated symbols (ambiguous black splotches on white paper, much like Rorschach tests) and constructing a common language out of them is the core challenge and the core mechanic of the game. Deciphering or combining

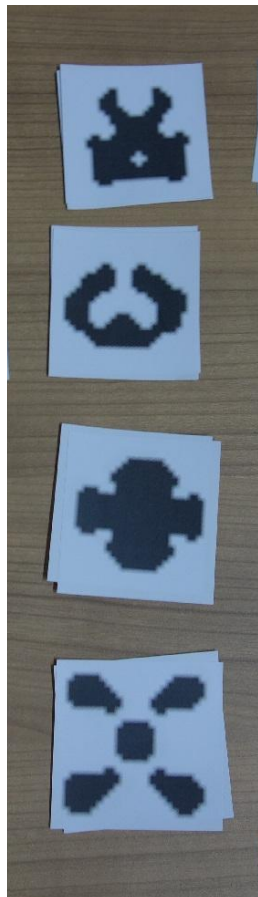


ambiguous images or semantics has been used in boardgames such as *Dixit*, *Codenames* or *Cards Against Humanity*, and forming a co-operative shared lexicon of abstract visuals is vital in *Mysterium* and *Codenames: Pictures*.

Alien Transmission uses generated abstract symbols instead of pre-authored ones, although admittedly the same could be accomplished by running the algorithm once (producing many symbols) and using a combination of 12 symbols in each game [This was how the playtest version for the Global Game Jam took place, as printing and cutting new symbols on the fly required too much downtime].

Notably, the generated black-and-white symbols are much more abstract than the evocative, colorful images of *Dixit* or *Codenames: Pictures* which are grounded in real-world representations (however dream-like). The more abstract nature of the symbols, the lack of real-world grounding, and the underlying algorithmic principles (smoothened curves due to cellular automata, emergent dots and splotches, symmetries and rotations) need to be incorporated into the players' association between perception (the visual symbol) and cognition (the underlying message). Generated content thus becomes part of the "grain" that players need to work with to win as it directly affects the core mechanic (transmitting a message).

Find the prototype cards, board and manual for *Alien Transmission* at <http://alientransmission.antoniosliapis.com>.



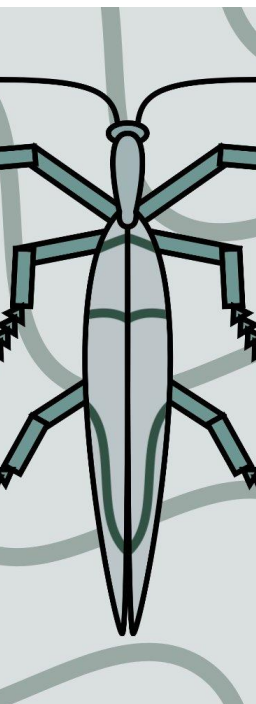
Bugs & Beetles

By Sabine Wieluch

<http://www.bleeptrack.de> | @bleeptrack

"So, yesterday I read the awesome article about generative art by @GalaxyKate and now I'm sitting at home, writing a script creating bugs. Like, literally bugs."

This was the tweet I posted earlier this year along with two images, containing a grid full of white bugs with black outlines. I would not have imagined getting such a huge and positive response.



After stumbling over GalaxyKate's great talk and blog post about generative art and her procedural flowers, I finally wanted to write a generator by myself. Something nice to look at, with easily recognizable shapes. But what exactly? Leaves? Too boring. Butterflies? The wing pattern might be too complicated to begin with. But beetles sounded right to me: They differ enough to be interesting and the 6 legs, 2 feelers and 2 wings make them easily recognizable.

The next evening I had two hours of spare time, because I waited for a conference call. So I opened Processing and began to think about how the beetles should look. Processing was still rather new to me, as I only used it for some simple generative art projects. But it supports Bezièr curves and I had lots of experience with them through extensive Inkscape usage.

So, I started with 5 points to generate my first shape: the center, the neck (where the head should be positioned later), the left and right shoulder points and the bottom. For these I could define the first parameters: the distances from the 4 points to the center (both shoulder distances are the same) and the Bezièr handle lengths of the neck and bottom points. To find the right parameter bounds, I just tried increasing or decreasing the values until the body started looking weird.



Now that I had the body, I continued similarly with the wings and feelers. The head only is an ellipse. And finally the legs: they consist of two rectangles and a row of triangles. And now the beetle shape is finished - yay! Over all, there are about 50 parameters.


Because of the great response to my tweet, I decided to keep going and give the bugs a colored version. Actually, this was the harder part. First: wing pattern. Two pattern were created very fast: one which consists of small, random placed little dots, the other one consists of concentric rings with the center at the wing center point. But, dots and rings were not exciting enough. So I decided to create a pattern made of wavy lines. Here I had lots of parameters to fiddle around with and a huge variety of wing pattern grew from this method. This is the part I am most proud of in retrospect.

“Now that I had the body, I continued similarly with the wings and feelers.”

Second: choosing colors. At first, I wanted to choose a random color and then color the different body parts in decreasing saturation. This looked nice, but again I thought it was not exciting enough. So I chose a second 'highlight' color for the wing. This looked better, but not really harmonic. Next, I calculated the gradient between the colors and used the in between colors for the non-wing parts. This looked a lot better, but because I choose the HSB color space, I often generated dark beetles and I wanted them to be more colorful. This happened, because in HSB color space, the saturation parameter goes from 'black' to 'bright color'. So if the random saturation value is low, the other values don't matter anymore: the result will be blackish. Gladly the resolution for this problem is simple: I switched to the HSL color space.

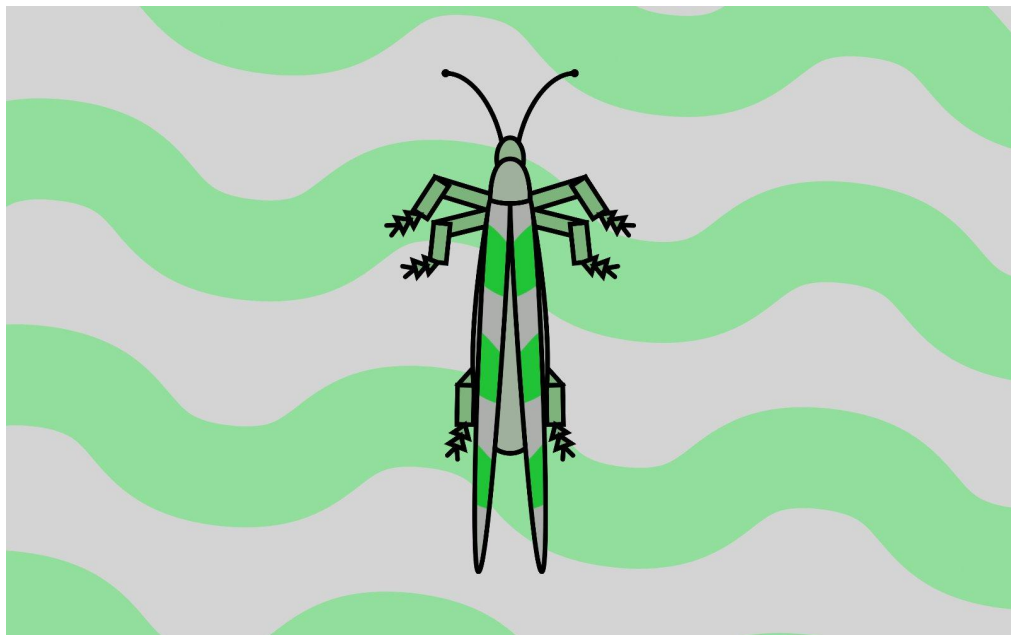
Now that the beetles can be generated, I started creating fun applications: first the twitter bot [@beetlesbot](#) that generates a new beetle every 6 hours and adds a nice generated name to the





bug. To create this bot, I had to port the whole code to p5.js (javascript version of Processing). And to make a nicer image, I added the wing pattern in low opacity to the background. This improved the bot images a lot! I made pen plotter plots, and therefore had to rewrite the whole code again in paper.js and find a new way to generate the wing pattern. I printed stickers, t-shirts and fabric to make shawls from it and I have lots of other ideas what to do with my new little friends.

If you want to play with the generator yourself, I recommend visiting beetles.bleeptrack.de where you can generate them yourself and play around with the parameter sliders.



On the Responsibility of Makers

By Gorm Lai

@gormlai | <http://www.gormlai.com>


This September I'll be starting a PhD at Goldsmiths in London as part of the IGGI programme. I am going to research algorithms that can empower more people to become character artists and animators through the use of procedural tools. When I excitedly tell my artist friends in the games industry about this I often get the reply :”So are you going to take my job away?” Even though it is meant as a friendly and funny remark, we all see how automated tools transform and change not only our industry, but nearly all parts of 21st century society.

As makers of something that makes something, we participate in bringing of some of that aforementioned change. Whether it is a procedural terrain generator to empower game developers to make more or higher quality content faster or someone constructing 3D printers to build houses, the change that we bring, affect the people in those businesses.

“While we can argue there are numerous advantages to 3D printing a house...”

While procedural tools can be incredibly empowering, they also take work that was previously done manually, and transforms it, so the work is either fully automated or will require a different set of skills to perform. For an easy to understand example, think about building a house. Building a house requires the combined skills of many people; architects, engineers, plumbers, bricklayers, carpenters, painters, etc., but if we can [3D print an entire livable house](#), then we might be able to leave out the bricklayer and carpenter and we'll instead need someone with a different set of skills to operate the 3D printing equipment. We will also need a modified supply chain for building materials, as instead of bricks, mortar and wood, we'll need a supply of 3D printing materials.

While we can argue there are numerous advantages to 3D printing a house (a machine can work 24-hour days and it is much cheaper overall) or a procedural tool for game developers (it might be



possible to knock up an interesting test level in minutes versus hours or even days), it is also possible that it will push some people out of the job market or at least require them to re-skill significantly to stay competitive.

Makers; whether they make medicine, cigarettes, guns, candy bars, self-driving cars or in our case, something that makes something, must take responsibility for their creation. If the creator of a gun or pack of cigarettes can be made accountable for its use, then so can a programmer for their software. That means if a piece of software allows someone to create something for the first time in their life because that software helps them to be creators or encourages a company to optimise their work processes, then both those cases bring change to society, and we are part responsible for that change.

We not only have to think about how people can best use our programs, but also the societal change that the software brings. If fewer artists are needed to make a game, what happens to those artists now? What about factory workers as we are able to [make cars in an increasingly automated way](#) or construction labour if [3D printed homes become more commonplace](#)? Is it always realistic to retrain people to other jobs? If yes, who pays for the training? If no, then what does society do with those previously employed people? Is it sometimes better not to invent something that improves our lives in certain areas or optimises production flow, if it has the potential to be too disruptive in other areas?

I have a my own set of answers for those questions, but this essay is not about that. The intent is to spark a discussion and make us realise our wider responsibility as makers of something that makes something.

A Web-based Node Image Editor

By Johan Rende

@Tokjos | <https://jrende.xyz>

The first procedurally generated piece I made was using photoshop. I felt that I lacked artistic skill, so I started with a cloud then added filter effects and blending modes until it looked good.

Using photoshop for that felt like overkill, so made a web application of that specific workflow. The result was filter stacker (<https://jrende.xyz/filterstacker/>). It was very limited in what you could do with it, but you could still create some nice images with it. Many years after, I noticed that I had started doing something similar with Blender: I would create a 2D square, and just add nodes to create interesting patterns on it.

I love starting ambitious projects of unclear usefulness, so I decided to create a web app of that workflow as well. <https://jrende.xyz/pattern> is the result. It is a node-based editor for creating 2D images, using WebGL for the rendering and ReactJS for the interface. It lacks a lot of features and polish, has a lot of quirks, and the only user the interface is friendly towards is myself.

“Right now, I’m not sure what direction to develop it in.”

Right now, I’m not sure what direction to develop it in. One is to improve the usability, make into a proper tool, like a web alternative to Substance Designer. The other path is to double down on its quirkiness, let it be purely for fun.

One “feature” is how gradients disregards aspect ratios. When you resize the browser, some parts move differently creating an animations of sorts. If I wanted it to be useful, I would squash that bug. But effect looks very cool, and I love how a bug can look beautiful.

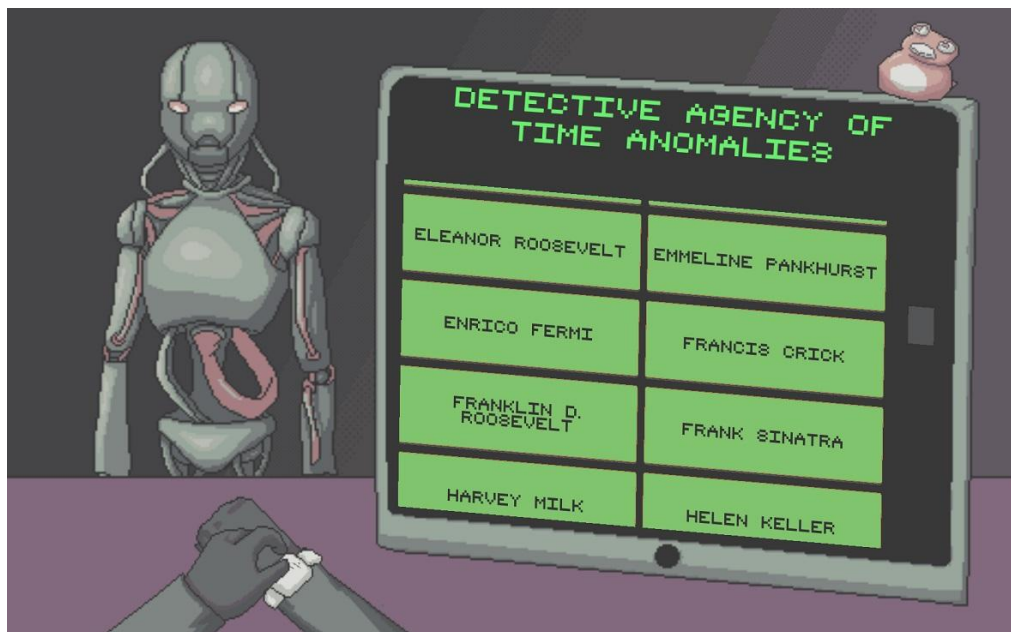
Another inspiration is glsandbox.com. In the future, I want my it to have the same invisible user system with the ability to fork images and track how popular ones are copied and modified in multiple generations.

Real-world Data as a Seed

By Antonios Liapis

<http://antoniosliapis.com> | @SentientDesigns

Living in the Information Age, we are surrounded by news, facts, propaganda, and advertisements. We are bombarded with information from TV screens in bars or from wall projections on the subway, and we can look up any type of obscure information on our phones within seconds (thus ruining trivia night). There are community efforts to fact-check and compile information in sites such as Wikipedia, PolitiFact or the Europeana collections. There are also far less noble efforts at using consumers' interaction data to target them with messages of all kinds. While games exist in all types of devices, we do not often think of games as sources of information --- least of all factual. However, games can take advantage of all this information available in repositories, websites and social media to create new ways of engaging players as well as disseminating information during gameplay.





Gabriella Barros, Mike Green, Julian Togelius and I designed a game which takes advantage of information on open data repositories and transforms it into an adventure game. The game is called ``DATA Agent'' and the player is an agent of the Detective Agency of Time Anomalies (DATA) tasked with solving a bizarre mystery. An assassin has traveled back in time and killed a famous person, masquerading as another famous person somehow related to the victim. Since the assassin does not know everything about the person they impersonate, the DATA Agent must find in a lineup of suspects which one does not have all their facts right. The correct facts about the suspects can be found by talking to other people in different cities. Finding the suspects themselves is no easy task either: the agent must talk to other people, read books and break into dark and locked places.

Most of the game mechanics in *DATA Agent* require the player to interact with real-world data, transformed into game elements such as locations, non-player characters, items, books and facts. The centerpiece is the murder victim, which is chosen by a designer before the game is generated. Around the victim, the generator starts by finding possible suspects among people with Wikipedia articles that share as many common attributes with the victim as possible (e.g. the same birth date or the same thesis advisor) but also have different attributes with other suspects. Once suspects are found, one of them is randomly chosen to be the culprit (the time-traveling doppelganger) and one of their facts is changed. Each suspect is linked back to the victim through a chain of entities in the Wikipedia knowledge ontology (essentially, finding the degrees of separation in Wikipedia:

https://en.wikipedia.org/wiki/Wikipedia:Six_degrees_of_Wikipedia); those links are transformed into in-game characters or books, placed in locations around the world based on their origin (e.g. their birthplace) or places found in the chain. Dialog with characters is generated based on templates to point to the next clue (unlocking a new character, item and/or location) but can also provide some



information about the character (such as their date of birth or subject). Finally, some puzzle elements are added by ``locking" some locations and ensuring that ``keys" can be found by the player (torches to unlock dark places, crowbars to unlock chained places).



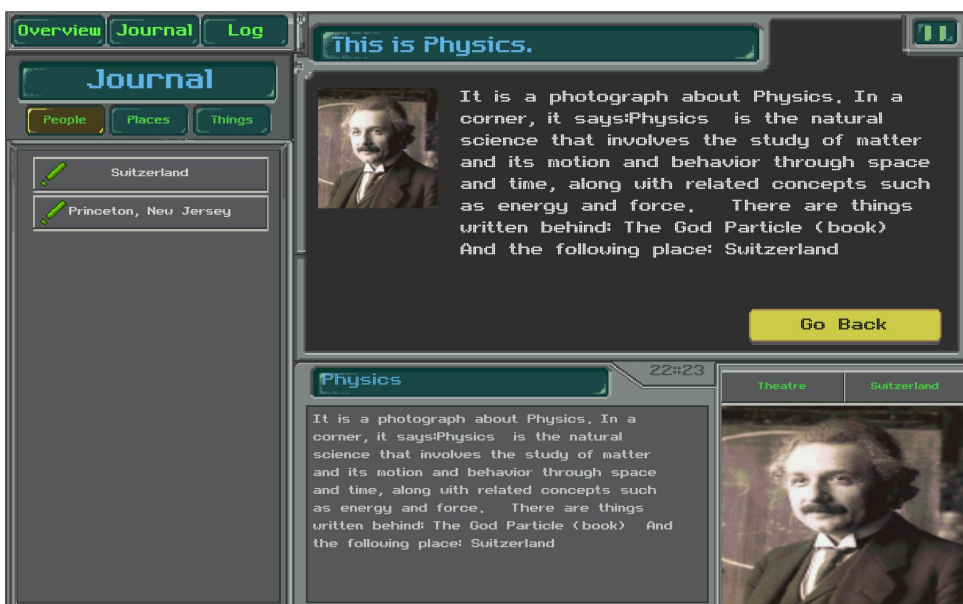
As expected, adventures generated by the DATA Agent algorithms do not always present a challenge --- or make sense. Suspects are usually connected to the victim, for example the suspects for the murder of Albert Einstein are all physicists. On the other hand, some of the falsehoods used to pinpoint the culprit can be obvious from common knowledge without playing the game. Some of the chains leading the player from the victim to the suspects also take bizarre

paths, for instance talking to Caligula and Khrushchev to find where the suspects of Frank Sinatra's murder live. Errors in the actual Wikipedia knowledge-base (DBpedia) can also lead to absurd results such as a non-player character named Argentina in the case of Louis Armstrong's murder (Argentina is of type ``person" in the DBpedia ontology, leading to this error). You can find out more bizarre outcomes, and play 99 generated adventure games by downloading *DATA Agent* from <https://champchampchamp.itch.io/data-agent>.



DATA Agent is far from perfect, but its aspirations are worth examining. Using real-world knowledge through open data repositories allows games to be more relevant and closer to the real world, recent news, or trending web searches. Trying to show the ``degrees of separation" in Wikipedia articles through a playful environment and a murder mystery narrative

(however absurd that is) requires both the generator and the player to rationalize why these links exist. *DATA Agent* transforms information of real-world people for the game, changing their time/place of death (for the victim) or blaming them for a murder they did not commit (the culprit); it explicitly states that history has been changed by a murder and a time-traveling doppelganger masquerading as a real person. However, in different scenarios (beyond murder mysteries, most likely) the real-world information could be kept intact within the game to allow players to explore real-world information, even for learning purposes. Using other sources of contemporary or opinion-laden snippets such as trending Twitter topics or activist websites can result in games rich in critique and conflicting viewpoints (but possibly sparse on facts, and thus unfit for learning). Games that highlight and expose data can even be used to debug or fact-check the original repositories, closing the feedback loop by correcting the knowledge-base that game assets were built from.



In short, *DATA Agent* is only one instance of how a semantically rich and narrative-heavy game built on real-world data can be used for entertainment, data visualization or critique.

Relevant readings:

Michael Cerny Green, Gabriella A. B. Barros, Antonios Liapis and Julian Togelius: "DATA Agent" in Proceedings of the 13th Conference on the Foundations of Digital Games, 2018.

Gabriella A. B. Barros, Michael Cerny Green, Antonios Liapis and Julian Togelius: "Data-driven Design: A Case for Maximalist Game Design," in Proceedings of the International Conference of Computational Creativity, 2018.

Website:

<https://champchampchamp.itch.io/data-agent>

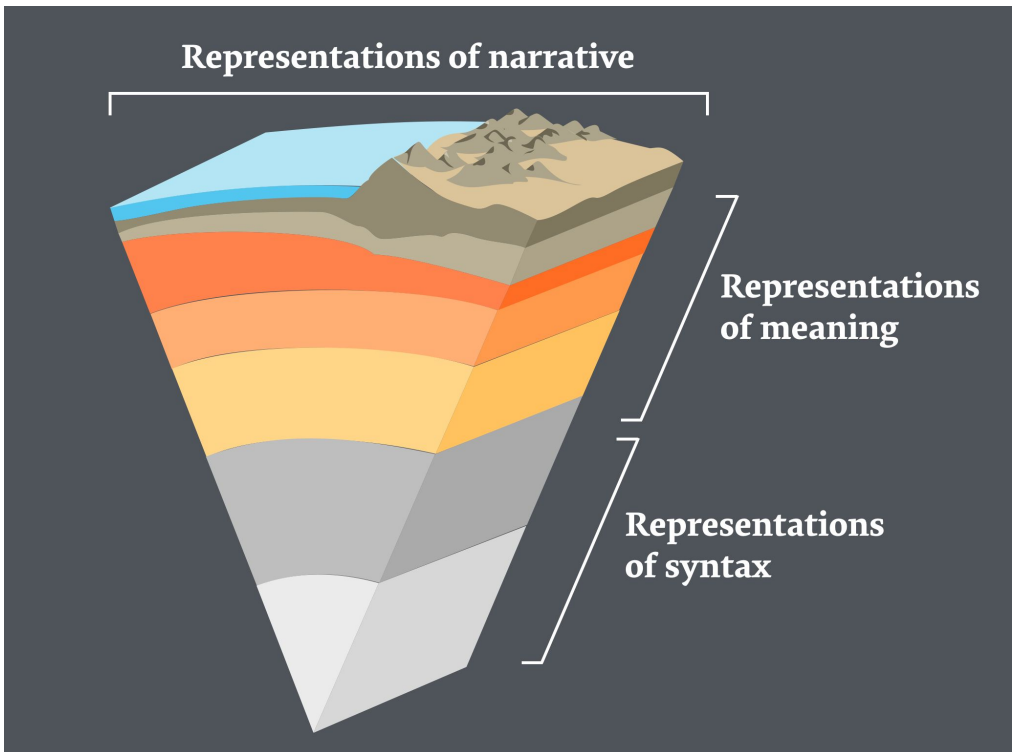



Representing Writing

By Mark Rickerby

<https://maetl.net>

There is no universal method of representing writing with computers beyond strings, which are linear sequences of characters. To create writing generators that operate at the level of narrative structures or use expressive, persuasive and literary elements of discourse, we need to think carefully about how we represent these latent aspects of the text. The way we represent these hidden structures is an important force in shaping what we can do with the generator.





One way to think about this problem is to look at generated writing as a series of layers or rock strata, moving from the low level atoms of language, from phonemes, letters and words, up to the higher level of meaning in sentences and paragraphs, and larger groupings like passages and chapters. Building on top of these layers leads us to representations of narrative, through elements of discourse, tone, characters, and plot structures. In concrete terms, we can look at all these components of writing as a tree formed out of the different levels of structure found in a text.

This model is immensely helpful in tackling the complexity of text generation. It makes it easier to understand the different methods of working with generated text by thinking about what level they apply to.

Another way of thinking about this model isn't so much about the levels themselves, more about the direction we're operating in — whether we're ascending or descending through the levels. Depending on which path we take, we end up with very different levels of control over the resulting text.

We could be starting with structure — plots, themes, tropes or narrative archetypes — and figuring out how to turn that structure into text by working down towards sentences. Or we could be starting with a large corpus of existing texts, working from the smallest pieces of syntax with sense and meaning being emergent.

It turns out these aren't just ways of classifying generative writing methods, but also AI philosophies that define the way we approach authorship.

The symbolic approach is about templates and top-down organisation, encoding our formalist ideas about the rules, patterns and constraints we want to apply to a piece of writing. It's more intentionally directed and crafted, but also potentially complex and

***“Right now, I’m
not sure what
direction to
develop it in.”***

requiring a lot of manual effort to get right.

The statistical approach involves turning text into data that we can operate on mathematically, and processing it using algorithms that aren't traditionally associated with natural language or text. Statistical methods usually avoid encoding rules about language or rules about narratives and plots. They treat text as a distribution of probabilities. It's more akin to musical sampling and remixing than anything associated with traditional writing.

Symbolic

'Top-down'

Authored rules, world models

Examples: grammars, graph rewriting, agent-based systems, goal-directed planning

Statistical

'Bottom-up'

Sampled from existing texts

Examples: n-grams, Markov chains, word vectors, recurrent neural networks

A lot of generative writing in recent years tends to cluster around one or other of these poles, exemplified by the weird and wonderful experimental works coming out of #NaNoGenMo and PROCJAM which are often made using a single specific method.

Every generative method is captivating and interesting by itself, but it's important to emphasise that the tradeoffs between these methods are not an either/or proposition. They all have different strengths and weaknesses, and operate at different levels of the writing process. Statistical methods require meticulous corpus selection and pruning to get right, while symbolic methods require a big investment in modelling and design.

There are fascinating possibilities for building writing machines composed by multiple generative methods, each addressing a specific level of language or narrative, with their inputs and outputs

feeding one another. Think of a Markov chain name generator embedded inside an expansion grammar that generates story fragments. Or a machine learning model that generates a plot which is filled in by templated sentences and entities from a world model.

As always, the needs of the story shape its generator, while the generator determines the limits of the story that can be told. In generative writing, the author is not effaced so much as working in collaboration with the system. Whether this involves curation, pruning, parameter tweaking or writing microcopy and sentence fragments is totally up to our imagination and creative vision for what we want to produce.

Narratives

Passages / Chapters

Paragraphs

Sentences

Words

Letters

Phonemes

Making a Fantasy Newsroom Bot

By Damien Crawford

<http://cannibalinteractive.itch.io> | @DeveloperDamien

I was working on a game during the summer that involved managing a guild and doing paperwork in order for your adventurer to be able to do anything. I was considering how to set up things like quests, local characters and shops; the local stock market for trade goods and such, and so on — but I wasn't sure what guidelines I wanted, or where I even wanted to start.

Then I heard that PROCJAM Summer '18 was about to start, and decided to use it for inspiration. Making a twitter bot (using cheapbotsdonequick.com) for an hour every day fit with my work schedule, and wouldn't be hard to have quick and tangible results. For the sort of information I wanted it to generate, the theme of a newsroom that reported on incidents and goings-on sounded interesting.

Thus began the concept for the Fantasy Newsroom Bot ([@RPGNewsroomBot](https://twitter.com/RPGNewsroomBot)). After doing some research on how other people put their bots together, I started looking at actual news shows and considering what to make it report on. One of the big goals was to have quest concepts, but I felt like that was a narrow topic and wanted to make it more like a proper newsroom report, so *that* category was renamed “Employment”. With that category, I could also talk about specific regions that are looking for certain classes to participate which worked really well with my original game's concept. Weather reports are common on the news, so I added that in too; since I'm working with potentially magical weather, I also made the reports mention what sorts of impact on trade and magic that the weather affected. More report topics like politics, crime, and celebrity news were added too.

From there I had to figure out a generation method for the reports that was coherent enough to sound like a proper news report, but still random enough to be interesting.

The first report I wrote was this:

*"In Weather today, #itempre.a# #weathertype# #weatherentry#,
#eleeffect# #percent#%."*

The news reports for weather always start with the same introduction; the terms in between the hashes are what categories the words it uses are pulled from. #itempre.a# pulls a word that makes sense for an item prefix (Flaming, Astral, Iron, etc.), as well as putting an a/an in front of it. #weathertype# picks a type of weather system, #weatherentry# picks a way for it to show up, #eleeffect# picks what elemental type is affected and in what way, and #percent#% picks a number and adds a percent sign behind it. A report generated with this setup looks like this:

*In Weather today, a patient tornado materialized, increasing Ice-type
skills' power by 75%.*

From there I focused on making three types of reports for each category for variety's sake, and adding more vocabulary for the bot to draw from. By the end of the jam it had about 16 different report types, and a lot of different ways for them to go.

As I continued to add to the bot it grew away from my original concept (to help with ideas for a game) and became its own project that I still add to from time to time. It now invents names of dungeons and magical diseases, and more is added every now and then. I do think that it would be interesting to make a game that actually uses it in some way, as some of the things that it comes up with are amazing and nothing I would ever have come up with on my own. However, I'm glad that this bot has become its own stand-alone creation, and I'm glad to have made it.

*"I'm glad that
this bot has
become its own
stand-alone
creation, and I'm
glad to have
made it."*

Procedural Generation in Twine

By Dan Cox

<http://videlais.com> | @videlais

Many people know Twine as a good tool for creating quick narrative games in HTML. With its story map view showing the connection between different sections it makes for a great first-time tool for people making games. Yet, what is often forgotten when thinking about passages, the internal units of Twine, as sections of a story is that they are also an internal form of storage.

Twine works by embedding its code and content into the same HTML document. When run by a browser, the code is read and transforms its elements into a story. As the player progresses through the story, the content of different passages are shown through reading from the elements in the HTML document.

This little detail about how Twine runs is important to remember because story formats like SugarCube, one of the three built-in story formats with the current version of Twine, come with functionality to retrieve the text content of passages while it is running. As HTML elements themselves, this passages content can be retrieved and even changed. And this makes them the perfect place to place data for a story within the story itself.

The string data type in JavaScript comes with a function to split a string into an array. Named `<String>.split()`, it takes a delimiter, something to mark one part of a string from another and returns an array of all of the items. Used with the function to get the contents of a passage, a long list of text data can be split into an array and then used within a story. Working just like Twine itself, the contents of passages can be inflated into memory as internal databases of names, places, or parts of things to drive more complex operations.

Through getting the contents of a passage, splitting it into an array, and then accessing its entries, it is possible to get a random entry to drive procedural generation systems. Combining these simple steps

with more complex rules, picking and combining things, for example, could drive even more complex integrations where one database of entries is picked at random and used as a seed for other operations and choices.

At the same time, Twine also comes with jQuery, a JavaScript library for navigating the representation of a HTML document as its Document-Object Model (DOM). Through using this library, the idea of using passages as storages can be taken to its logical extreme: passages, because they are represented as elements in a HTML document, can be added and changed while the story is running. Using the ability of JavaScript to change the properties of objects in memory, the story format SugarCube can be hacked through adding HTML elements using jQuery and then adding new internal passage referenced to its internal representation. Through combining jQuery with SugarCube functionality, it is possible to have a story build itself in real-time, reading and writing new code into HTML elements and linking to it internally through its JavaScript representation. Having the potential of building off of passages as forms of data and tables, it is possible to create a bare HTML document of different collections of text where code re-writes the HTML document in memory, creating new connections, parts, and generating new connections based on the code reading and rewriting itself through JavaScript, jQuery, and knowledge of how Twine handles its own internal storage.

“Through using this library, the idea of using passages as storages can be taken to its logical extreme...”

While Twine can be thought of as a tool to create more heavily-authored stories, it can also be used to create complex procedurally generated content as well. Through recognizing that a Twine story is a HTML document, it can be treated as both something to read and something to write. Combining knowledge of how Twine works with more advanced JavaScript usage, Twine can be morphed into creating stories that can write themselves, building off of rules, databases of text, and the inherit properties of hypertexts to connect to sections within themselves.

A Year of Daily Generative Sketches

By Martin O'Leary

<http://mewo2.com> | [@mewo2](https://twitter.com/mewo2)

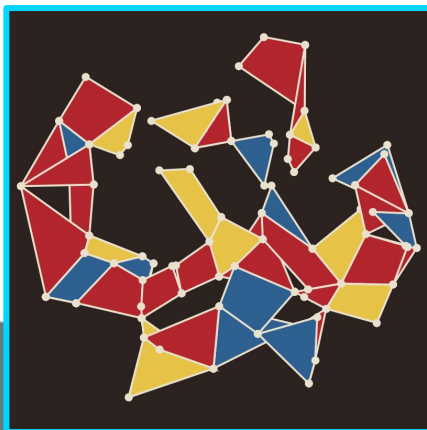
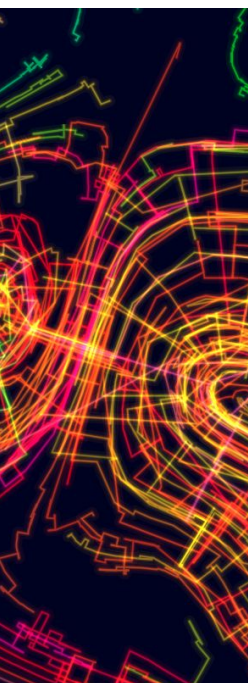
On September 13th 2017, I started a new project.

Every day since then, I've sat down and produced a "sketch", a small piece of generative art. Each piece takes the form of an 800x800 pixel image, which I post on a dedicated Twitter account ([@mewo2sketches](https://twitter.com/mewo2sketches)).

Many artists, across all kinds of artistic disciplines, do daily projects. There are lots of benefits to this kind of practice: it keeps your mind in a creative place, it lets you try new things in a low-risk way, and over time you build up an impressive portfolio of work. At the same time, having a side project like this can be really helpful psychologically when you're stuck on a long-term project, or when you don't have time to work on anything bigger.

I find generative art is particularly suited to this kind of daily work. If I only have a few minutes one day, then I can tweak a few parameters from yesterday's work, and create a whole new image with only a little bit of work. If I have several hours, I can experiment with new algorithms, and make something new and exciting. Either way, I have something new to show before I go to bed at night.

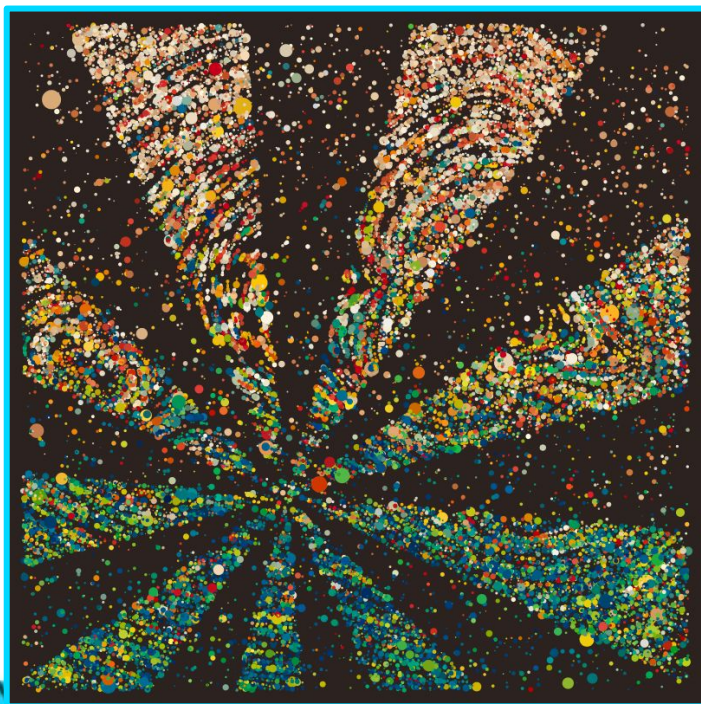
I want to give some advice for if you're planning to do this yourself.



Start Now, While You're Excited

When I first had the idea to do daily sketches, my thought was that it would be a great New Year's Resolution. The problem was that it was still only September, so I'd have to wait four months to get started, which was kind of discouraging. It took me a few days to realise that I could start any time I wanted, and that waiting would just mean my enthusiasm would drain away.

The first week or two, I was really jazzed about the whole idea, and I put a lot of work into each piece. Then, as always happens, I hit a patch where I had less motivation. Having already built up a little stack of sketches, I didn't feel as much pressure on the newer ones. I knew that some would be good, some would be bad, and that each individual piece was less important than the project as a whole.





Make Your Own Rules

When I started, I set myself a few rules:

- * One piece per day, no more, no less
- * Ignore the clock, a day is between waking up and going to sleep
- * I can stop any time I want, but I can't skip days
- * Everything is code, no external data sources
- * No text, just graphics

These are my rules though, not yours! Constraints like these are really important and useful, but they have to be something you choose for yourself.

Work In Public

It can be a bit rough, putting stuff online which you haven't spent a long time on, which maybe isn't as polished as you might like. There's a temptation to keep this sort of thing private. Obviously I can't stop you, if that's what you choose to do, but I'd really encourage you to post your work publicly.

Putting your work online has three main roles. It gives you positive feedback - those likes and comments will be a great boost when you're feeling down about the project. It also gives you accountability - you won't be as tempted to skip a day when you think other people are going to notice. Finally, it means that you build up a public record of your work, which you can point people at when they ask.

Procedurally Generated Spellbook Sprites

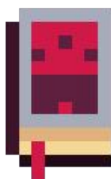
By Spencer Egart

<http://zeno.itch.io>

These sprites were generated to be used as spellbooks in a roguelike game (*Lore of Rune Era*). The process for their generation is pretty straightforward — a series of image layers, each of which can have one or more possible sprite, are colored and drawn on top of each other. This allows for variation in the shape and details of the sprite — some of the books have different bookmarks or clasps, for example, and vary in color.

Each book is also given an icon for the cover, which is generated simply as a purely random (50% chance to be on or off) grid of pixels mirrored over the X and/or Y axis. Over the small size used here — just 5 x 7 pixels — this results in intentional-looking icons.

In the context of the game itself, the book sprites can be controlled to give a visual indication of their contents, so that a book of fire spells could be primarily red and orange, a book of healing spells could be white and gold, etc. My current work here is to extend this type of sprite generation to other types of items such as weapons, armor, and potions.



Procedural Generation, Variety and Reproducibility

By Davide Ciacco

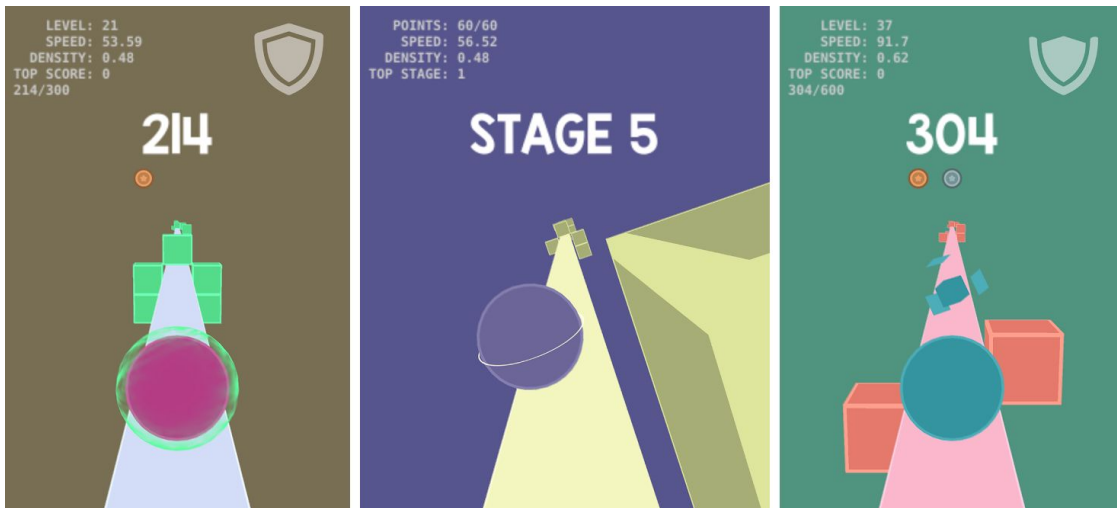
<https://ciaccodavi.de/> | @ciaccodavide

Most of the time I hear people around me talking about procedural generation only as an algorithmic alteration, or generation, that grants some things a wide variety — and that is a very important feature!

If used wisely, procedural generation can provide our games with art, music and events which work together to make them always engaging, interesting and replayable. But, this "variety", if leading to excessive randomness, can be unsuitable for certain situations.

In some games with procedurally generated levels, especially if a leaderboard is present, every player should be playing the same exact level so that the score is fair against other players results. In these cases procedural generation should grant both variety and reproducibility.

A couple of real examples: in my game "Idle Duels" there are procedurally generated armors and swords (in pixel art) and the enemies encounters happen in the same order for every player, they have the same statistics so that the position of a player in the leaderboard is fair.





The same thing happens in my game "Pepidox", a simple game where you have to avoid obstacles. All the levels are procedurally generated, but the colors, the obstacles positions and properties like speed or density are not random! Every player will have to face the same challenge that each other has.

I'm currently working on a game that features procedurally generated music, I will probably choose a certain seed to generate the main menu music because, while variety is good, sometimes is better to reproduce the same thing for everybody to make it recognizable.

The reproducibility aspect of procedural generation can also be very useful in multiplayer online games, where some events and contents can be procedurally generated reliably on all the player's devices (the clients) connected to a certain match, without the need to exchange data with the server.



Letting Go of Realism

By Guerric Haché

<https://guerric-hache.itch.io/> | @GarrickWinter

The pitch is easy! Generate a complex world with dynamic societies, like what you'd find in *Endless Legend*, then play in it as a single character like you would in *Skyrim*.

That pitch is the last remotely easy thing about the idea. But, after generating worlds as a hobby for years, I've found some approaches slow me down more than others. One I think I've possibly just understood is the deceptively sensible pursuit of realism.

Intricate world generators have long amazed me — from plate tectonics and erosion to wind flow and temperature gradients, people put enormously impressive work into these things, and for many years I tried similar approaches to generating complex, interesting worlds for hypothetical RPG-like games.

But I was never satisfied with the results. Despite multiple layers of simulation, they felt generic, with ecological gradients smeared across large distances and no clear sense of distinct regions or places. Each little generator eventually suffocated under increasingly complex code that was difficult to maintain or debug, and I moved on. I enjoyed building generators, but why didn't the results feel right?

I figured it out after a brief return to *World of Warcraft* in 2017. I hadn't played since before Cataclysm radically altered Azeroth, so the leveling experience was wholly different. I spent a lot of time contemplating my emotional reaction to these changes to a world I had once loved, and something struck me.

Azeroth stops even pretending to realistic geology and climatology on any scale larger than what the player generally encounters on-screen. Zones are ludicrously segmented by rectangular mountain ranges, humidity and precipitation and rivers are slapped

wherever convenient, and there's very little gradation between ecological zones. And as I traverse that world, I don't care — I love it. What matters is the player-scale; the zone's unique combination of colours, art, music, narrative, and themes. Together, they create a sense of place and make the experience of exploring a new zone or town a distinct, compelling event.

Or did, before Cataclysm. The non-realistic, characterful world design of old Azeroth only became obvious to me as I parsed my viscerally negative reactions to world design post-Cataclysm. Cataclysm, it turns out, hacked away at zones' individuality and grafted on generic lava and fire, generic Alliance-versus-Horde narratives, and generic militarized characters and architecture — the same generic stuff across multiple zones. That last part is key — aesthetic and narrative bleed between zones increased dramatically, making individual places feel less, well, individual.

What did this teach me about generating worlds? Two things. First, worry less about simulating reality; at player-scale, global realism counts for little. A RPG player primarily experiences individual places, not orbital world maps, so it's individual places that need to be most interesting.

Second, worry more about differences and distinctions between places. Gradual, irregular gradients across dozens of kilometers may be realistic, but traversing them on foot won't yield many memorable moments of suddenly being somewhere else. Contrast is a powerful tool for creating those moments, and for geography I've found two helpful principles for contrast: deliberate consistency within a region, and deliberate differences between regions.

My next generator takes these to heart, and I'm already happier and more confident in the results than I ever was designing simulationist generators. It divides the world into arbitrary zones, simulates climate and geology on a zone-level only, then works on reinforcing

***"What did this
teach me about
generating
worlds?"***

those two principles - internal consistency, external heterogeneity. Zone climate and geology are consistently applied within the zone, while the generator tries to ensure adjacent zones are different from one another.

I've achieved compelling results with - here's the best news for a hobbyist - far less work than any previous approach. Any point in one region will be significantly different to a point in another region, and the code is easy to work with since there are no granular simulations. In fact, the workflow and results are so compelling I'm already moving to the next step - feeding these worlds into a functional 3D game with a controllable avatar.

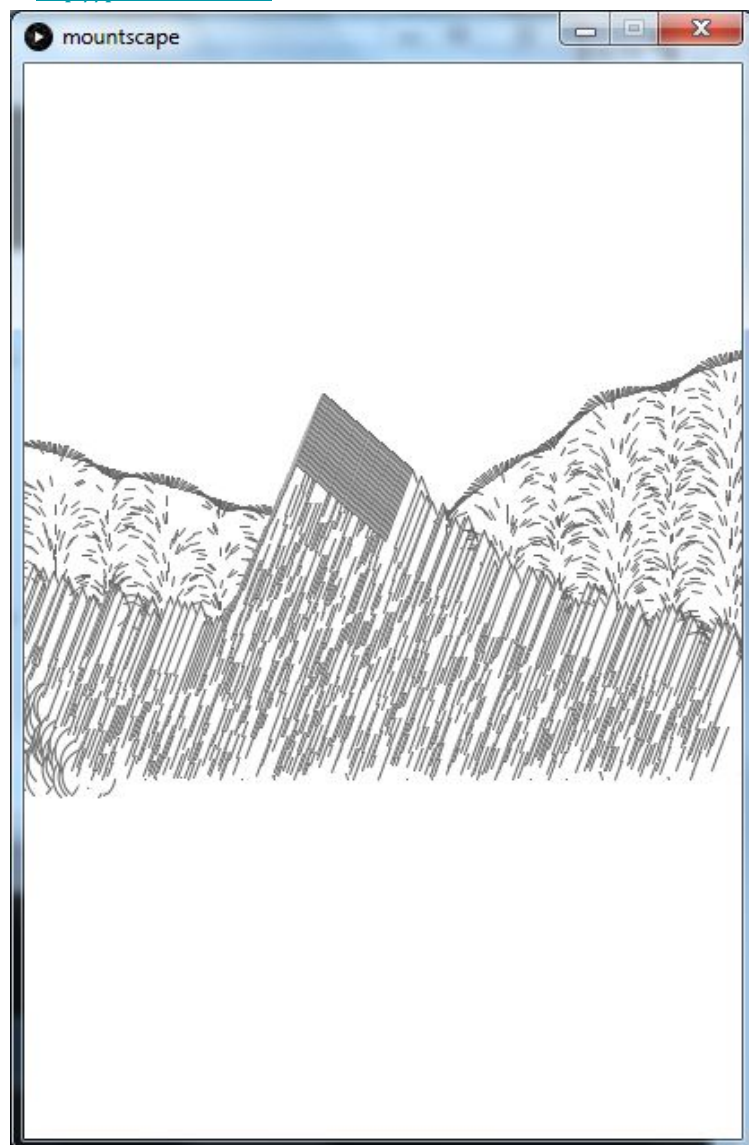
Realism as a concept is often lionized, and its benefits can seem obvious from the orbital view of many map generators, but realism often isn't the quality that actually brings us joy in games. Vanilla *WoW*'s world with its unique zones resonated with me, regardless of its rampant affronts against physical geography, because it created places and moments that made an aesthetic impact.

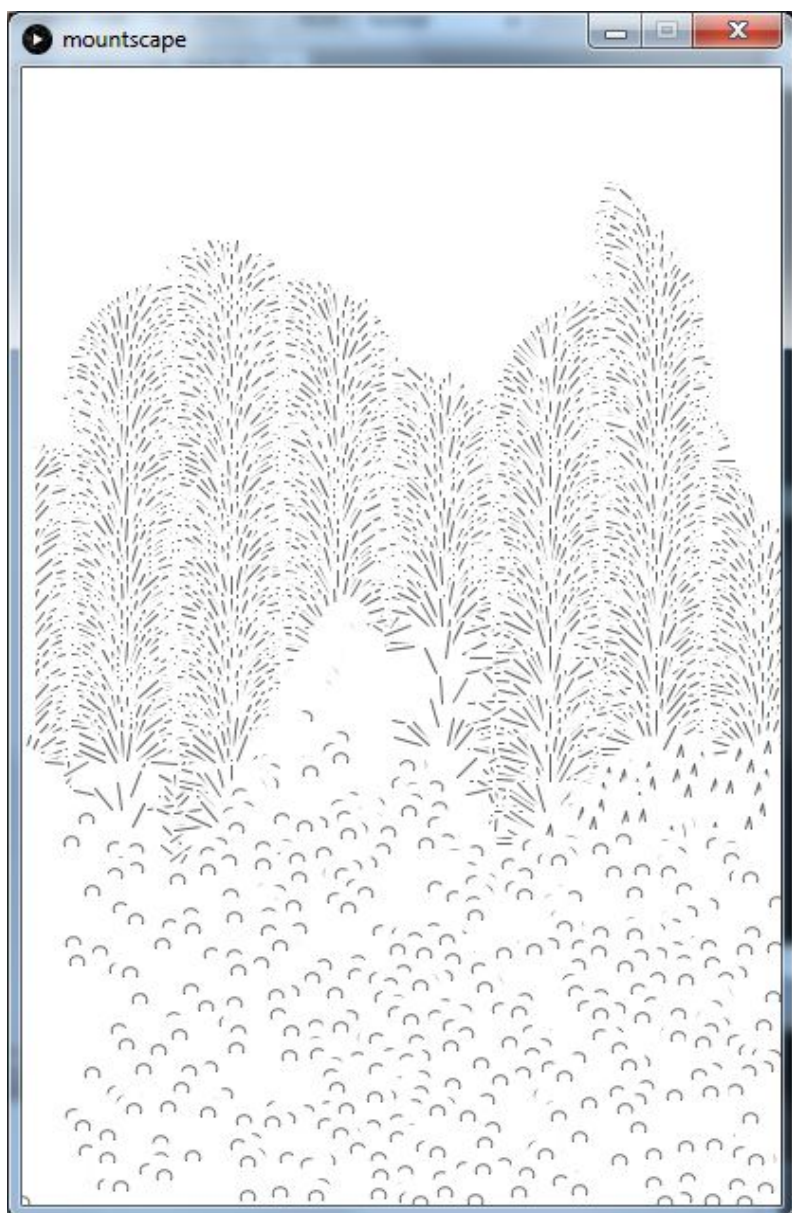
So that's the idea I'd like to leave you with. In addition to real-world counterparts to thing you're generating, take some time to consider the gameplay experience you want your procedural content to support. Play the experience you want to generate, then ask yourself what makes that experience special, and try to grasp those qualities algorithmically. You might just find a new angle for tackling your own outrageous procgen ideas.

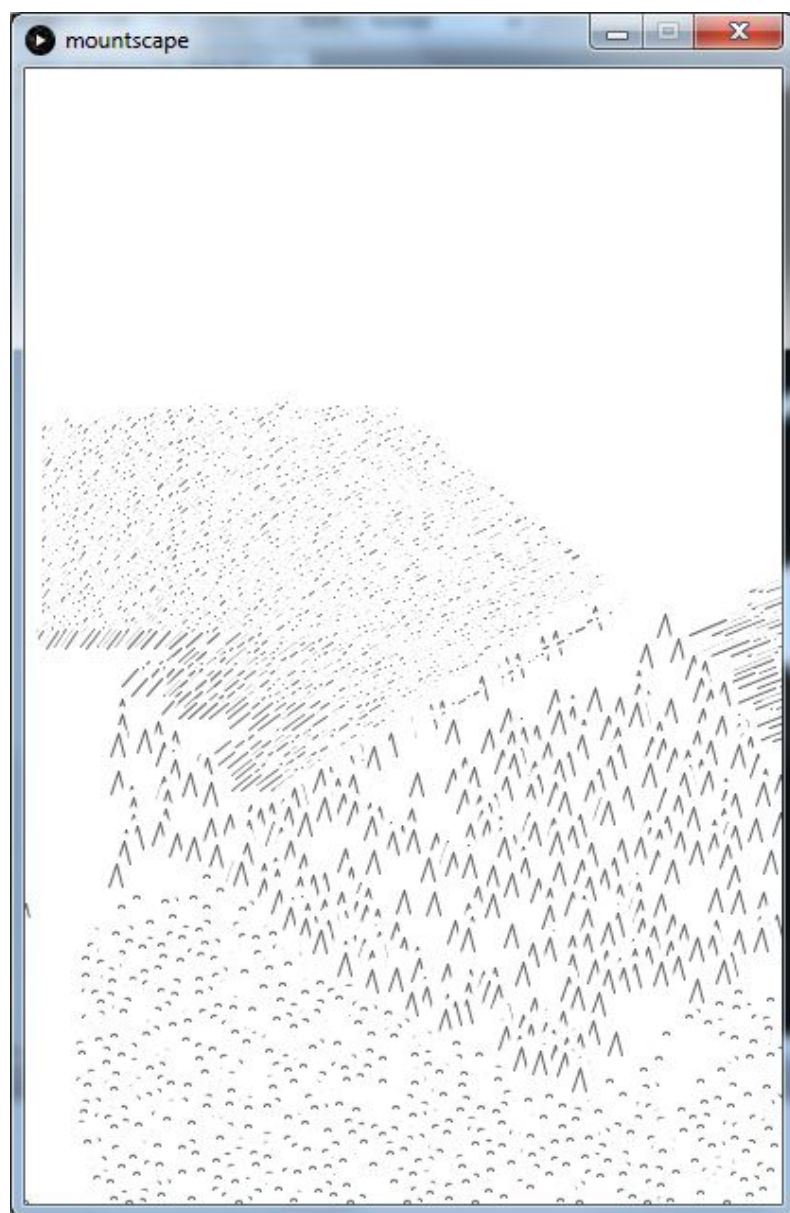
Mountscapes

By Pol Clarissou

<http://polclarissou.com>







Mere Juxtaposition - On Generative Fiction

By Jasmine Otto

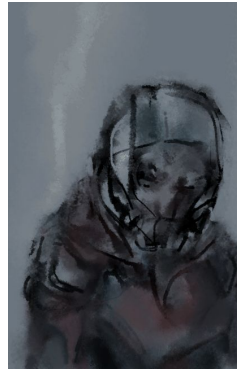
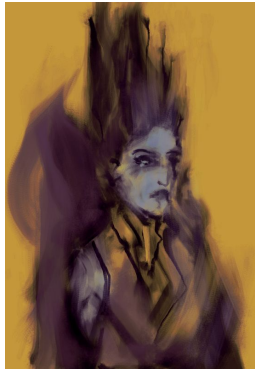
<http://jazztap.github.io/exul-mater/> | @jatazak

Liner notes on a lacunic genre fiction, initiated this Summer Projam.

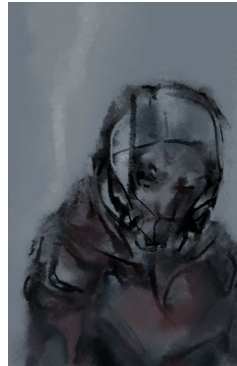
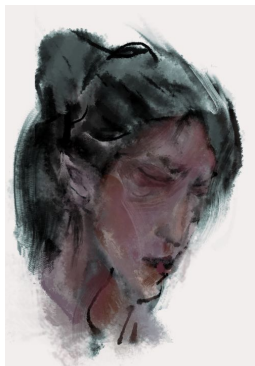
<https://jazztap.github.io/exul-mater/>

mere juxtaposition

Consider the narrative coloring of an illustration by its adjacencies. Allow those gaps [1] which expand this space of interpretation.



the hanged woman, a saint | the devil, her apprentice



the magician, unmasked | the devil, in her armor

There is plot already. The reader authors it without conscious effort.

We would extend this technique, derived by designers from the Kuleshov montage. Leaning on mythology lets this demo be concise, but setting-knowledge in general is expensive. Our approach is a demake into text.

'Exul Mater' is a proof of approach, in the form of a fanfiction (er, 'myth-making practice'), using the prompt-generation techniques described here.

Act III - Transcript

...now that you have rank. How did you gain that?

By denying the enemy their agent, at the same time faking your own death.

She fell from the sky like a star. Shouldn't have let you on the transport shuttle, to be fair.

Let's examine the tempering effect of The High Priestess (aligned w/ winter, the harsh aspect of proving) compared with Judgement (aligned w/ moth, the fitful aspect of unknowing).

knock

"You know, savages aren't meant to be carved from marble." Her hand slips quickly from your too-heated grasp. She'd pass for Imperial - not that all citizens are pale.

"Did they tell you our hearts were cold, devoid of human feeling?" Idyll doesn't even blame you. "That if we created, it was only for practical purpose? I am sorry."

grail

As the world is divided into evening and dawn, so we gave the wandering star two names, and it became divided. They will know you by your faces, which are your kith and kin, as you are theirs in turn.

These are the tokens, which stock our proper nouns with their initial state. Across all tokens, the entire cast is represented. In both renditions, the same passage (knock/grail) will appear, with different adjacencies.

knock / winter / grail

The master of a particular Remnant engine, who revealed the ruin inside a city, had sacrificed only the distinction between persons and resources. Therefore his assertion, his sum of efforts, was only the existence of naked power - which pond scum also prove.

The white phosphorus reminds Meletus of their failed apprentice. The Huntress taught you to handle the the devil's element because of its use in smoke grenades, not for its effect as an incendiary. Oh well.

You speak to me in riddles, tight knots of self-recrimination. You fear, or hope, absent lovers' regrets will turn to blame.

You turn from me, and drape the curtains' embrace across your shoulders. You cannot take mine, or break the illusion.

You are like your mother more than you know. You insist upon your judgement.

The High King's eldest apprentice, your mother's lover, fell in the wake of the coup. He'd taken to regret too well, perhaps. He burned with abhorrent martyrdom.

His crimes were not treacherous, compared against hers. His redemption should have been the High King's curse broken. So what if he would not be cold? He should have burnt himself numb.

One foot in the Order, which could not see itself hollow,
and one foot out, where its strain could not be exploited.

Soon the actual will match the shattered image.
Rahel uses Idyll for access to Caelum, and her throne.

knock / moth / grail

When she made us shatter(ed by) the collapsing planet, a deep well in physical spacetime, our general knew then that to remain present, sensate, would be to die or be destroyed.

Traitor our marshal who broke her own master upon her duty,
Betrayed and Betrayer who brought our general back to slaughter her own academy.

You memorized your mother's weaknesses while young.

You speak to me in riddles, tight knots of self-recrimination. You fear,
or hope, absent lovers' regrets will turn to blame.

You turn from me, and drape the curtains' embrace across your shoulders. You cannot take mine, or break the illusion.

You speak of the woman you loved, the woman you destroyed. Did you join her on the tower steps? Do you regret what she became?

"We must be ashamed to mangle, upon the same old battle lines."
Her husband did say this much to me. Does your cohort enjoy their

limited grasp of history? You may forget so many useless details, and never reify them again.

ASK YOUR father whom he could not keep you safe from, that HE BEGGED for you to be made civilized and civilizer. I think he understood this goal.

"Seeking ourselves, we'd be as killers, or healers, Rahel!
Which role should I have led, so you'd need not?"

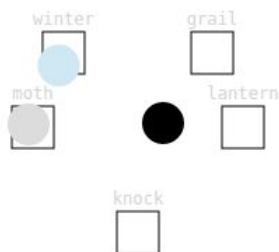
But her child lies beneath Idyll's tree whose roots are cables,
whose boughs are dendrites.

Production

The Parrigues Tarot suite [2] introduces certain desiderata of generative text. (Similar to Calvino's Memos, except for the respective outliers Mushroom, and quickness.) As our rubric:

Beeswax (multiplicity)

A human generates all natural language content, ****and**** enough world state to indicate 'saddle points' where scenes **should go**. Other methods of annotating the relationship-graph and other features of world-state are much slower.



Salt (exactitude)

Scenes correspond to the 28 possible pairs ($8 \times 7 / 2$) of 8 tokens. In practice, three ‘contexts’ of 5 tokens each (with 0 or 1 unique to each context) sufficed to gate repetition of scenes, and produce movement between alternative world-states (as by the technique of AUs).

	lantern	moth	grail	edge	heart	winter	forge	knock
lantern								
moth								
grail								
edge								
heart								
winter		not love	love					
forge		not hate	hate					
knock		not death	death					

Egg (lightness)

We prefer each scene within drabble-length (≤ 100 words), to reveal secrets / images (as units of character) one or two at a time. Prose of shorter length exerts greater juxtapositional force. Occasionally the third token alters one paragraph, where the ‘triangle closure’ can be helped along.

Venom (visibility)

Referring to character, a conceit for audience investment. The mere co-existence of tropes in one well-motivated being is interesting, but also, irreducible to a single relationship or interaction.

Mushroom (proliferation)

Most visible in plot, a useful lie to reveal the characters' relations. All causality is ascribed (in the absence of reproducibility – and social graphs, even outside of heightened narratives, are full of edge cases). We implement a ‘rearrange’ slider, so the order of scenes is wobbly, and the reader may deliberately author this bit.

tokens

With established proper nouns (including the narrator!), even trivial sentences may take on special significance. This will help us land the endings. (It is useful to block in a few endings early on: the richness of prompts partly arises from accounting for their changes.)

Proper nouns point to characters: human individuals, organizations, other emergent phenomena, & even useful tools. Character **facets** are created by multiple readings, their traversal of fragmented roles.

The desiderata of characters are **aspects**, which are a basis for the subspace of thematic space. I've used a pre-existing set pertaining to immortality, originally in order to understand their off-dichotomy juxtapositions. Evocative aspects can inform many characters at once.

endings

Since Exul Mater has three 'factions', each token gives points of loyalty toward one or two of them. Those kind(s) of point which predominate determine the ending.

A 'loyal' ending yields an appropriate confrontation, roughly. Split loyalties are feasible, e.g. resulting in multiple destroyed enemies.

Our ending texts are short - otherwise, they overwhelm the scenes, - and not very literal. Better that they surface themes that emerged in the writing, & wouldn't fit in the title.

scenes

Here is the bulk of the writing. Take each pair of tokens to index a scene. Every relationship between characters has a tone, many of which are expressed succinctly in songs. Use songs to prototype

scenes! Use them to find verbs. Do not work from a single word, but a striking image.

Do not associate songs to *tokens*. If you can't write out a sentence, it's not yet concrete. If you can, put it in your scene. It's hard to experience power if you don't smash anything, and other feelings need similar reification.

Δ closure

Once scenes are drafted, we can exploit the nature of pairs to find a saddle point (the moment before 'climactic resolution') in every scene.

Given five slots, the first two tokens determine a scene A/B. The third token determines the remaining scenes {A/X, B/X} where X varies over {C,D,E}, framing A/B in three possible ways.

By this *triangle closure*, the recombinatorial fiction maximizes its density of saddle points. How many ways can one scene be read? Whence the players' stakes, their motives? Mere context.

The appropriate text editor reveals this information during writing. I would encourage experimentation.

beta.observablehq.com/@jazztap/ritual

[1] catacalypto.itch.io/on-lacunae

[2] emshort.blog/2018/06/26/parrigues-tarot-draft

[3] www.ice-bound.com

Intelligent Middle-Level Game Control

By Amin Babadi

@donamin



We propose the concept of intelligent middle-level game control, which lies on a continuum of control abstraction levels between the following two dual opposites: 1) high-level control that translates player's simple commands into complex actions (such as pressing Space key for jumping), and 2) low-level control which simulates real-life complexities by directly manipulating, e.g., joint rotations of the character as it is done in the runner game QWOP.

We posit that various novel control abstractions can be explored using recent advances in movement intelligence of game characters. We demonstrate this through design and evaluation of a novel 2-player martial arts game prototype.

In this game, each player guides a simulated humanoid character by clicking and dragging body parts. This defines the cost function for an online continuous control algorithm that executes the requested movement. Our control algorithm uses Covariance Matrix Adaptation Evolution Strategy (CMA-ES) in a rolling horizon manner with custom population seeding techniques. Our playtesting data indicates that intelligent middle-level control results in producing novel and innovative gameplay without frustrating interface complexities.

Superorganism Art

By Genetic Moo

@GeneticMoo | <http://www.geneticmoo.com>

A superorganism is a big organism made up of smaller organisms. Superorganism art is a big piece of art made up of smaller pieces of art.

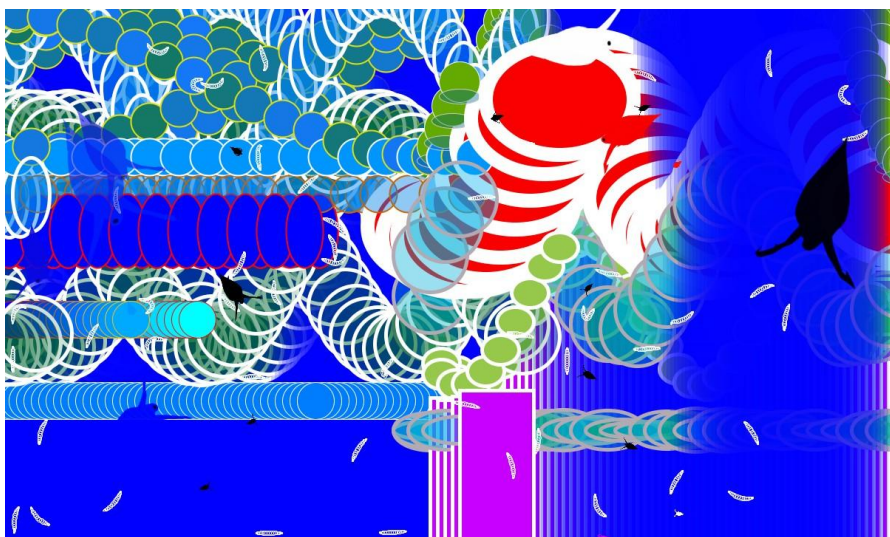
With readily available and easy to use digital tools, superorganism art dramatically expresses the powers of collaboration, communication, critical thinking and creativity.



We recently created a multiple projection piece with the participation of over two hundred 15-17 year olds as part of the UK's National Citizens Scheme which aims to give teenagers a range of skills to make them 'unstoppable' — like the way an ant colony conquers all before it, and an ant colony is a great example of a superorganism in nature. The workshops were run over 4 days in Canterbury, Kent and then we combined all the results together into a series of generative programs to be projected on a huge scale at a

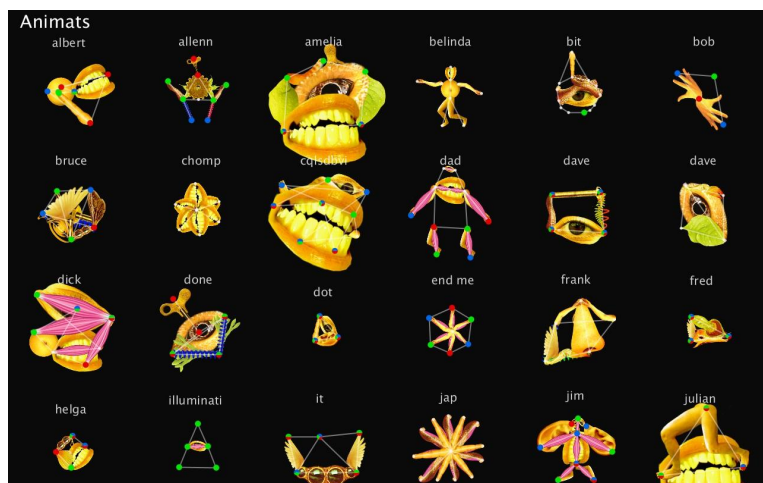
local art festival.

But where did this superorganism art idea come from?



For years we had been making interactive art with computers and sensors and projecting these works into group shows. We started thinking about how these artworks would interact when there was nobody in the space, as there is nothing sadder than interactive art with no audience. We were told to be careful that our projections and sounds did not leak over into another artist's personal space. And then we had enough of that — we decided to reverse this rule and worked with artists who deliberately bleed into and over each other. Artworks were designed to interact with artworks. Then you put the audience in the middle and see what happens. Nobody knows what will happen! Digital Art is particularly good at this — the outputs change in response to the inputs according to collections of behavioural algorithms and sensors positioned by the artists. Combine enough of this together and the resulting chaos can be thrilling to behold — and can develop a life of its own.

We started to think how we could get non-artists involved and designed a series of workshops where beginners could learn some creative coding and make small generative animations — unique to each person. Using the Processing language meant these could be taught almost instantaneously — in the space of an hour anyone can create something substantial with a rich variety of output over time. The next stage is to copy each person's code into a larger container program and then run all the code snippets at once in a shared space. By varying each workshop thematically we further enrich the output. For the Canterbury workshops we used the four elements EARTH, WATER, AIR and FIRE as starting points for people to experiment with.

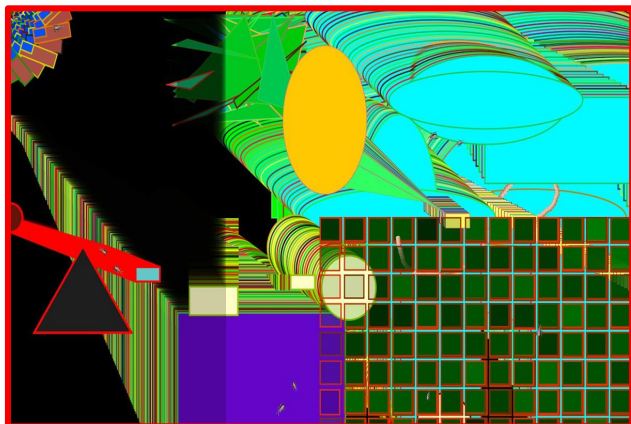


Now as this is a Genetic Moo project we added in some digital creature design (ANIMATS) and some of our own artificial life forms (amongst them CORAL and TERMITES) to occupy the shared space and respond to the ever changing animations and a superorganism is born.

So, we have designed a system which allows people to make small parts which come together, combine, and emerge into a larger whole.

All the while teaching how easy it is to make little steps in coding which can be combined into bigger steps which can make a dynamic and life like artwork. The possibilities are endless and we shall leave you with a young H.G.Wells's speculations on Early Experiments in Co-operation:

"The recent work undertaken by physiologists to investigate the behaviour of the peculiar corpuscles in the body, the phagocytes, lends colour to this vision. These strange unities wander through the body, here engorging bacteria, and there crowding at an inflamed spot or absorbing an obsolete structure. They have an appearance of far more initiative and freedom than a factory hand in the body politic. It is as startling and grotesque as it is scientifically true, that man is an aggregate of amoeboid individuals in a higher unity, and that such higher unities as may be reasonably likened to man, the Polyzoa individuals and the Ascidians, have united again into yet higher individual unities, and that, therefore, there is no impossibility in science that in the future men should not coalesce into similar unified aggregates. There can be no doubt that such phenomena as the now almost forgotten Siamese twins and double-headed monstrosities are tentative experiments on the part of Nature towards a 'colonial' grouping."



not_constantinople

By Seth Alter

@subaltrngames

not_constantinople creates place-names from a blend of cultures, e.g. "three parts English, two parts Catalan, one part Assyrian". In other words, you can create names that don't sound like generic high fantasy using it.

I created not_constantinople as part of a much larger worldbuilding project; I wanted to populate a map with town names but I was struggling to imagine them on my own (especially if I was less familiar with the cultures in question). When I later found out that sci fi writers were interested in it, I cleaned it up and released it for free.

The challenge in creating it was finding a big enough corpus of place names sorted by culture. I eventually converted Crusader Kings II's data files into a (heavily modified) json file.

not_constantinople has a few more planned features, in particular: a web app version, and a personal name generator ("not_constantine").

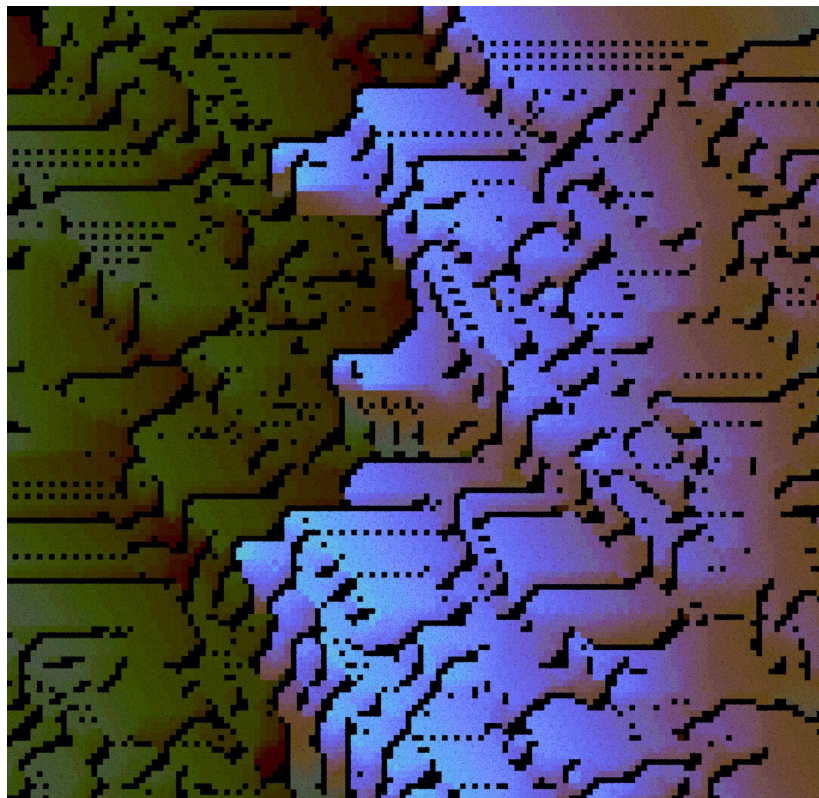
Download not_constantinople here:
https://github.com/subaltrngames/Not_Constantinople

Nos Falaises

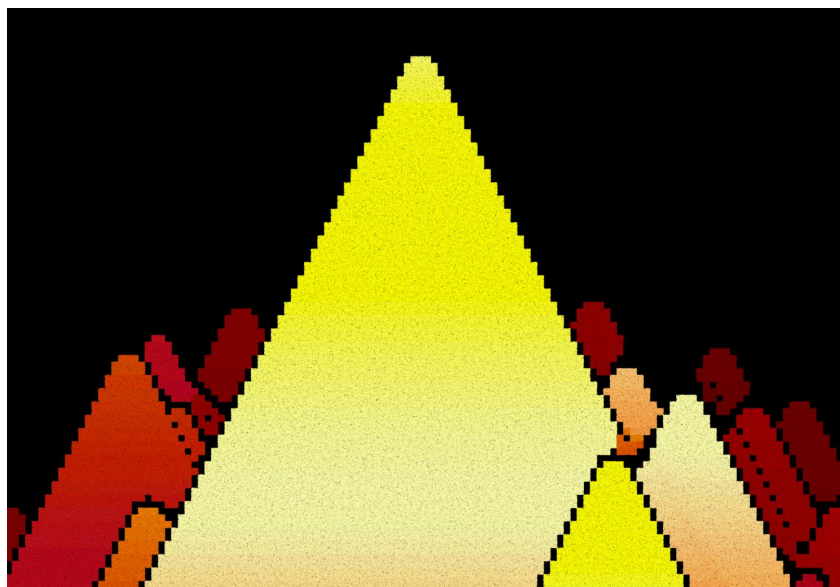
By Guillaume Pelletier-Auger

<https://pelletierauger.github.io/> | @pelletierAuger

*"A presitation
of comic
books"*



NOUS APERÇÛMES enfin le
canyon aux alentours
duquel devait se trouver le
temple de Marguerite.



Au matin, nous gravâmes les
escarpements malgré nos jambes
meurtries et nos yeux mi-clos.

Arrivés au sommet, nous
prîmes quelques difficiles
respirations et franchîmes
finalement les portes de la
clinique externe de psychiatrie
de l'hôpital Jean-Talon.

More: <https://pelletierauger.github.io/Nos-Falaises-Presentation/>

Minus World: Generative Game Reviews from a Parallel Universe

By Max Kreminski

<https://mkremins.github.io> | @maxkreminski

[Submitter's Note: A few days ago I was digging through my lab's extensive board game collection and found a strange-looking gaming magazine, apparently published by an organization called the "Tlön Society for the Advancement of the Ludic Quasisciences", sandwiched between a couple of the boxes. Upon closer inspection, I found it to be full of reviews of games that don't actually seem to exist. I don't yet know quite what to make of this discovery, but several of the reviews describe games that make use of procedural generation in very interesting ways, so in lieu of a "real" submission, I've elected to reproduce a few excerpts from the magazine's numbered "Best of 2018" list here.]

34. YAWP (Fern Buckley)

Much like veteran designer Fern Buckley's previous three games, YAWP is a grueling QWIPlike that starts out tough and only gets tougher as you gradually come to terms with the true nature of the challenge.

To begin with, you've got the usual awkward mapping of keyboard buttons to individual muscles; this time it's the muscles that control a human vocal tract. Your first utterances will horrify the ear, and even once you start to get the hang of the controls, the real challenge still awaits: the NPCs around you, whose own utterances are easily mistaken for gibberish at first, are actually speaking an entire procedurally generated language that you have to learn to speak yourself (through the same awkward button-mashing interface) in order to progress. Seriously, this game's subtitle should be "Developmental Linguistics Simulator".

As a concept, YAWP is brilliant. As a game... well, I ragequit about 3 hours in. Consider yourself warned.

17. Gutter (Fake Palindromes)

Gutter is a game about truth. Something terrible (no one's quite sure what) befell the world about 700 years ago. Civilization bounced back well enough, but the paucity of surviving documents from Back Then has left you, the historian-protagonist, facing down a mystery of literally apocalyptic proportions with basically no real evidence to go on. One day, some enterprising archaeologist discovers a huge cache of well-preserved newspaper comic strips from the years immediately preceding the Event. Now, the race is on to pick through the strips and piece together a coherent explanation of What The Heck Happened. If the public finds your story convincing, you might just be able to establish yourself as a historical authority and save your faltering academic career.

Gutter is massively replayable. The world's history, including both the exact nature of the disaster and the comic strips themselves, is procedurally generated on a per-playthrough basis. Different cartoonists have different styles and senses of humor; they might be aggressively political or focused on the everyday, relatively impartial or horrendously prone to bias. As you struggle to deduce what real historical events the strips might be referring to, you have to take all of this into account.

At times, sifting through the myriad layers of indirect reference to scrape together some scrambled impression of the truth feels like trying to reconstruct the events that set off the latest discourse on Squawkbox (or whatever social burrow you prefer) from vague subsquawks alone. It can make for some frustrating gameplay – but the moment of triumph when you finally assemble a story that fits, one that can withstand the scrutiny of your colleagues and resonate with the public imagination, makes all the struggle feel worthwhile.

6. Ruin Value (Softwary)

The premise of *Ruin Value* is deceptively simple. As the personally appointed Chief Architect of a tyrannical and image-obsessed dictator, you've been tasked with securing the legacy of the present regime. You are to oversee the construction of great wonders, majestic structures capable of outlasting even the death of your entire civilization by hundreds or thousands of years. To assist you in this task, the resources of an entire procedurally generated empire have been placed at your disposal.

As you build, your employer watches closely over your shoulder, frequently touring construction sites in person and offering his feedback on your progress. These visits are a constant source of stress, compounded by the unreliability of your supply chain (resource shipments are often delayed due to constant fighting on the empire's fringes) and the difficulty of managing a veritable army of laborers (many of whom are not offering their services voluntarily, and will take any opportunity to stir up trouble).

Nevertheless, you make progress. Until one day the empire falls, and you're suddenly booted out of the game for six whole real-world months.

When (if) you return, the camera has shifted to first-person, and the intricate management interface is gone. All that's left for you to do is walk around the remains of what you built – weathered and worn by millennia of simulated time – and reflect on the value of ruins.

Oddifier: RPG Character Sheets

By Paul McCann

@polm23 | <https://dampfkraft.com>

These are character sheets for the game "Into the Odd", a tabletop RPG set in a weird 19th century. The vast majority of the photos come from the Internet Archive Books Project Flickr account, and the images were composited with ImageMagick.



A character sheet for Arlo Kardos. The background image is a portrait of a man with a mustache and a turban, wearing a pearl necklace. A blue ink drawing of a ferret is overlaid on the left side of the portrait. The sheet contains the following information:

NAME			
Arlo Kardos			
STR	DEX	WIL	HP
10	15	12	3
EQUIPMENT			
Musket (d8)			
Ferret			
Mutt			
NOTES			
No sense of smell			



A character sheet for Ivy Brouwer. The background image is a portrait of a woman in a dark dress. A blue ink drawing of a ferret is overlaid on the left side of the portrait. The sheet contains the following information:

NAME			
Ivy Brouwer			
STR	DEX	WIL	HP
9	11	11	5
EQUIPMENT			
Musket (d8)			
Hand drill			
Shovel			
NOTES			
Can't swim			



NAME			
Horace Csordás			
STR	DEX	WIL	HP
8	10	14	5
EQUIPMENT			
Sword (d6)			
Shield			
Drum			
NOTES			
Missing an eye			



NAME			
Harper Fejes			
STR	DEX	WIL	HP
11	8	6	5
EQUIPMENT			
Pistol (d6)			
Fancy Hat			
Fire oil			
NOTES			
Iron limb			

All Dinosaurs are Great and Small

By Genetic Moo

@THISISDINOSAUR | @thetinySAURS

You may remember my project *THE DINOSAUR GENERATOR* from 2017's issue of *Seeds*. It's a big project, and has kept me busy for a couple of years, and will probably keep me busy for many more. I like where it's going, and don't want to rush it or make compromises, but sometimes you just need to get something finished.



With that in mind, I chose a project for 2017's Proc Jam that was, in many ways, very similar, but also entirely different. The end goal is still to generate dinosaurs, but that's all it does. You ask it to give you a tiny dinosaur (aka a tinySAUR), and it gives you a picture of a tiny dinosaur; you can't direct anything about it (e.g. what colour should it be, how aggressive should it look), and you don't know anything about the final result (e.g. where it lived, what it ate, how it moved). Expedience was the main motivation behind every decision, that is, what would allow me to produce pictures of tiny dinosaurs as quickly and easily as possible?

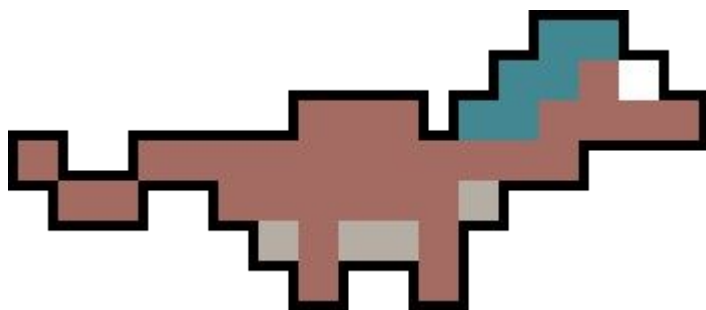
The generator is split up in to 8 sub generators, for different types of dinosaurs:

- Ceratopsians
- Pachycephalosaurs
- Ankylosaurs
- Stegosaurs
- Iguanodonts
- Sauropods

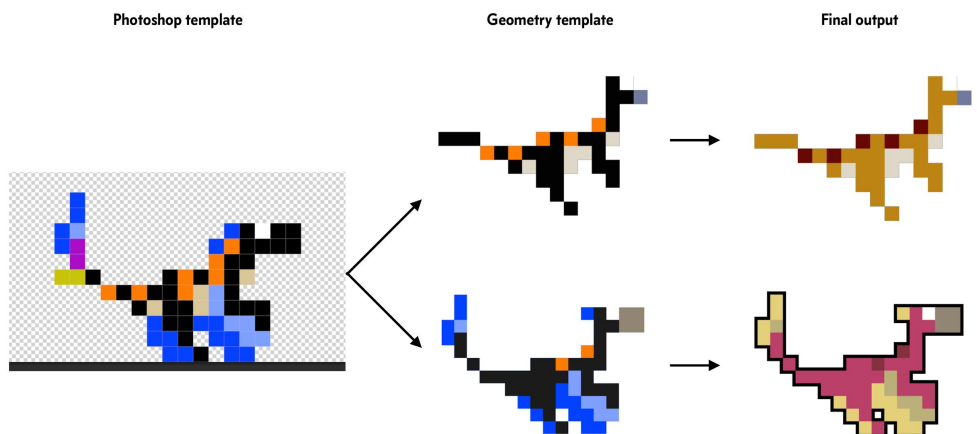
Large Theropods

Small Theropods

Each one uses the same basic approach, with generation split into two phases: the first step produces a colour-coded geometry template, and the second colours it. For the geometry step, each sub generator has a photoshop file, with layers sorted into groups for each part of the anatomy (e.g. a group of tail options), which the generator then may pick zero or one of the layers in that group (e.g. a tail is compulsory, a horn isn't). Some groups are also given a range of possible spatial offsets (e.g. some horns can be lower down and a position will be randomly selected). This structure is also leveraged to do different kinds of modifications, such as instead of selecting one from a number of body options, there may be a number of layers that can be added to modify the appearance of a the base body. There are also some conditional rules, such as if the top ceratops horn is too low or too big, it's not allowed to have a lower one. The photoshop files are colour coded, with separate colours for main bodies, horns, spines, eyes, feathers, and a number of colours specific only to the theropod generators, so that feathers can be removed, lengthened, and coloured with different patterns, allowing the generator to use the same files for both more retro scaley theropods, and theropods that are practically birds, like archaeopteryx.



The output of the geometry step is also colour-coded in the same way, although it may have decided to remove horns or spines, or decided that certain categories of feathers should actually be part of the body, amongst other things. The colourer then decides how to colour each template colour (e.g. what the body colour should be, if each different feather template colour should be the same, or totally different, or if there should be a gradation). The colourer uses many different colour palettes, with different chances of selecting from each (and some chance of having any colour combination). For example, one might be more likely to make the dinosaurs have nice earth tones, and another might make them look more like tropical birds.



An example of the photoshop templates, after particular body parts have been selected. Black is the body colour, the blue colours represent different layers of feathers, the yellow is body colour that should only exist if the dinosaur doesn't have feathers, whereas the purple is body colour that should only exist if it does have feathers. Another colour not shown here (turquoise), represents body colour on non feathered dinosaurs, and the first layer of feathers on feathered dinosaurs.

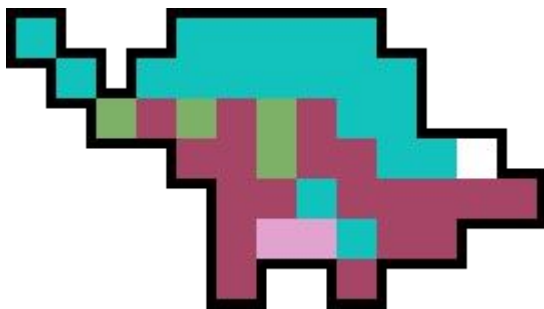
Example output of the geometry step. You can see how the tail corresponds to the template differently in the case of the feathered and non feathered dinosaurs.

Final output from the colourer. Optionally a black outline is applied to the output, which the twitter bot uses.



Depending on the sub-generator, many other techniques are used to add variation, such as with the sauropod generator, where any sauropod has a 50% chance of having some sort of back frill; this is then further modified by deleting a random amount of frill from the back to the middle of the sauropod. There is also a chance of removing the head portion of the frill (as long as the resulting frill would still be reasonably large). These rules were specifically selected to best match the sauropod references I could find. In this way, I could have the option of any sauropod having a frill without having to duplicate every possible neck, body, and tail option, and instead adding the frill to every possible one, and deleting it programmatically if it's not required. Similar techniques are used for the texturing, where masks for under-shadow or stripes are applied to the whole template, then parts are deleted selectively. There are simply too many special case rules like these for each sub generator to list them all in detail, from things like the variety of iguanodon head texturing rules, to the ceraptops head patterns.

If you want to see the results for yourself, you can follow the twitter bot [@thetinySAURS](https://twitter.com/thetinySAURS), or there's a server that lets you generate them yourself: <https://the-tinysaur-generator.herokuapp.com>



Story Generation by Algorithms: A First Attempt at a Digital Storyteller

By Adam Riddle

<https://realtimeriddle.itch.io/story-generator> | @RealtimeRiddle

Everyone gather around to hear the story of Little Red Riding Hood:

[Granny called Red, and asked her to bring some food.

Red goes to the woods.

Red makes it to Granny's.

Red leaves Granny's and goes to the woods.

Red returned home.

The End]

Wow, that was boring. Let us try again:

[Granny called Red, and asked her to bring some food.

Red goes to the woods.

Red makes it to Granny's.

Red leaves Granny's and goes to the woods.

Red runs into the wolf and introduces herself.

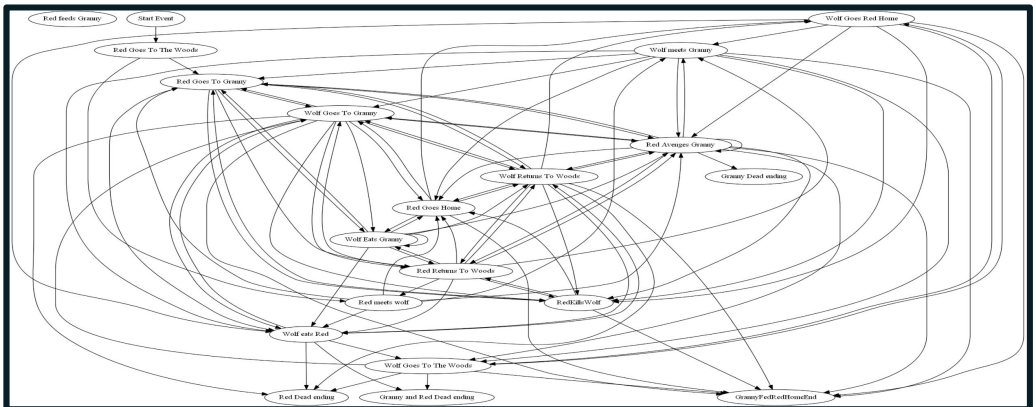
Red hits the wolf.

Red tells the wolf about Granny.

The Wolf eats Red.

The Wolf gets to Granny's.

The End]



Well that was darker than expected but that is the great thing about getting a story generated by a computer. You never know what you are going to get. These particular stories were generated by an algorithm that chooses the order of events to a story, and even though the text is the same for any particular event, the results can be very entertaining.

The Algorithm

This story generator focuses on changes in a story world caused by possible story events. The generation has two important components that consist of the story world and story events. The story events have two important components called preconditions and effects. Preconditions are needed in the story world to use the event, and effects are what changes in the story world after the event is used. The story world is a collection of variables and values that make up the current state of the story.

“This story generator focuses on changes in a story world caused by possible story events.”

The algorithm itself is simple. First, to initialize the generation, an initial story event is chosen. The initial story world is set to match the event’s preconditions.

Next, the generation starts with the story world being changed by the chosen event’s effects, overwriting existing variables if necessary. Then, a list of potential events is chosen by checking every event’s preconditions against the story world. An event is rejected if there is a variable in an event’s preconditions that is not contained in, or whose value does not match, the story world. Otherwise the event is added into a list. Once a list of potential events is compiled, an event from the list is chosen at random. This process is repeated until an “theEnd” variable in the story world is set to “true” by an event’s effects.

Problems

In the brainstorming stage this generator was more complex. At first there were three levels of generation planned. This algorithm would



have been the same process explained above with each layer, though events used for a different purpose depending on the layer. Once an event was chosen for a layer, another one would not be chosen until the layer below was complete.

The first, and highest, layer had events that controlled the overall direction of the story. The event's effects on this layer would have set a goal for the story world that would be reached with the next layer of generation.

The second layer would generate how the story progressed from one story event chosen by the first layer to another. The effects of this layer would act as a goal for the next, identical to how the first layer chose the goal for the second layer.

The final layer would have been used to put together tags for a grammar, an abstract structured language, to generate text. This layer's effects would actually have changed the story world.

The purpose for constructing the algorithm in this manner would have been to create novel stories that would surprise the author of the generator. The three layers of generation would have abstracted the process enough to create an unpredictable, but enjoyable, story.

However, such a complicated algorithm proved to be unnecessary as this goal ended up being largely accomplished with the simpler algorithm that I did implement without magnifying the numerous problems of the generator, such as authoring.

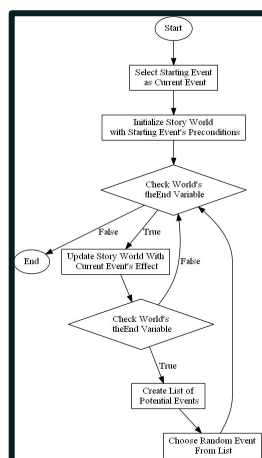
The authoring process can be both tedious and challenging as the author must manually ensure each event's preconditions and effects maintain the desired logic of the story. In essence, for each story event, every possible variable in the story world needs to be considered. This calls attention to how complicated the authoring of a story through events can become. In a way, that is intentional as

the complexity enables the generation of unexpected stories; however, this structure also makes crafting a desired possibility space of potential stories by hand nearly impossible. Just looking at a graph of a small story makes this evident.

The Future

To fix the flaws with the current iteration, better data structures need to be considered. At the highest level of generation, a finite state machine could be used to determine the overarching flow of the story. A planning algorithm should also be implemented to have an easier way of computationally generating and testing the network of possible events. The planning algorithm will also help remove needless repetition during generation and enable the use of multiple agents working simultaneously.

The next iteration will move from Python to Unity. This change will focus on easily authoring new stories using graphical user interfaces, enabling text generation after the events of a story have been decided, and procedurally generating interactivity to immerse readers into the generated story. My hope is that the next iteration will be able to generate consistently enjoyable stories with unforeseen outcomes.



Islands Are Just Mountains Up To Their Necks in Ocean: Part 1

By Scott Turner

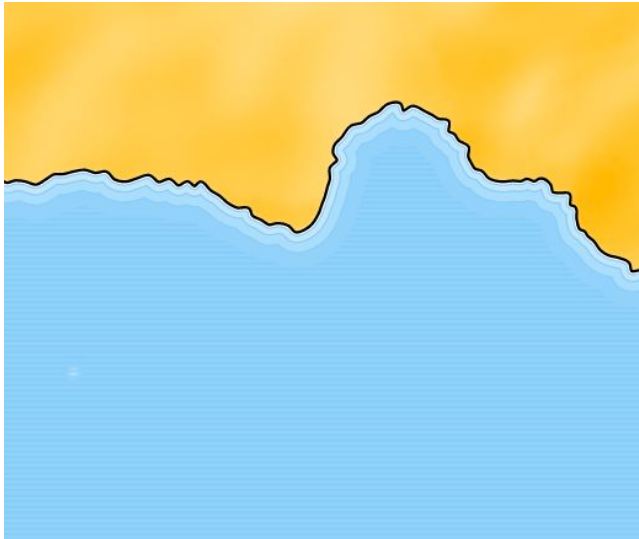
Generating terrain is a very common procedural generation exercise – many people tackle this as a first step to learning procedural generation, and many games use procedural generation for some or all of their terrain. Most terrain generation approaches you'll find on the Internet use Perlin or Simplex noise. With careful selection of parameters, you can use noise to generate realistic-looking mountains and interesting land shapes.

One problem with using noise in this way is that there is little control over the end result. If you are generating the terrain for some specific purpose, it's hard to guarantee that the terrain will suit your purpose. For example, if you need terrain with many small islands, it's difficult to be certain that noise will generate that. This might be the time that noise generates rolling hills and deep valleys instead.

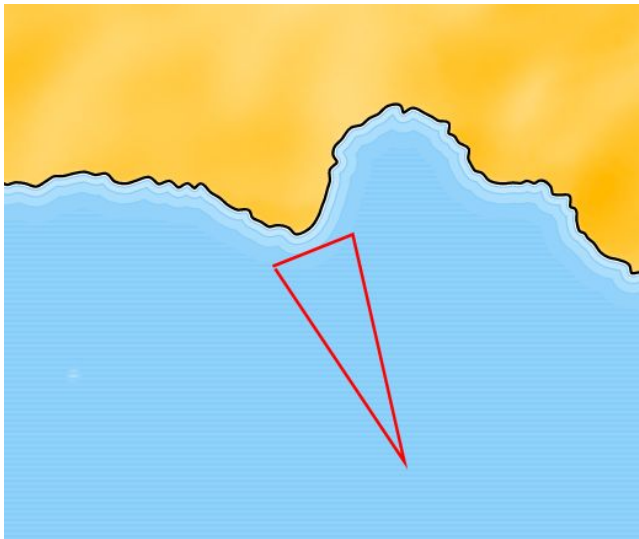
For the past few years, I've been working on a fantasy map generator called. *DRAGONS ABOUND* uses noise for terrain generation, but it also has a number of methods to procedurally generate various terrains that are more intentional. This enables *DRAGONS ABOUND* to create interesting maps by combining terrains in specific ways. In this article I'll talk about how *DRAGONS ABOUND* generates an archipelago of small islands, but you can read about a number of other methods on the [DRAGONS ABOUND](#) blog.

Some islands — like the [Hawaiian islands](#) — are formed by volcanoes rising from isolated spots on the sea floor. Other islands — like the [Caribbean islands](#) — are formed by the same sort of plate tectonics that create mountain ranges. It's just that the action happens where the ocean is deep enough to cover up most of the mountains. Only the tops of the "mountain range" rise out of the sea, forming a chain of islands along the edge of the tectonic plate. This suggests creating an island chain by using the same [process used to create mountain ranges](#) but putting them in an ocean area and sinking them so that only the tops are visible.

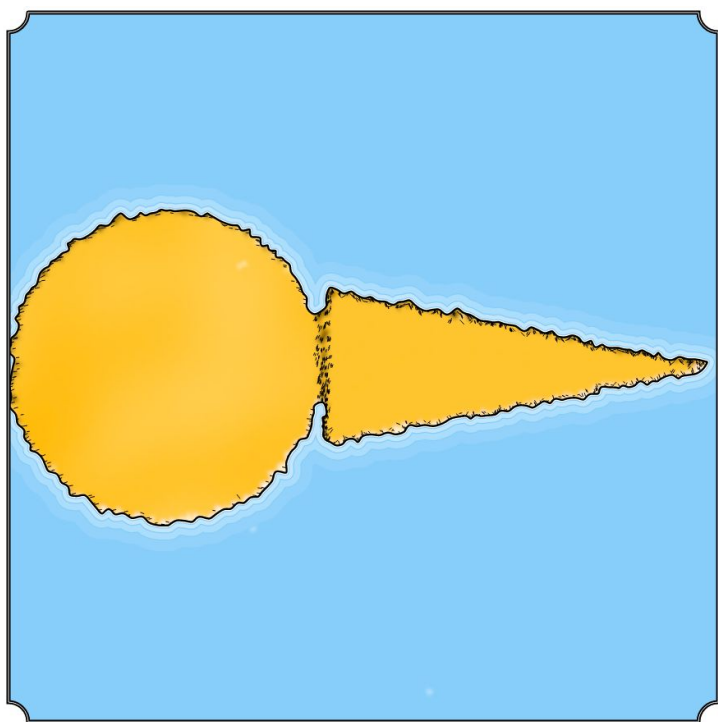
While this would work to create free-standing ocean islands like the Caribbean islands, in this article I'm going to use this technique to create island chains that extend off the end of a peninsula, like the Florida Keys:



I want to add some islands in roughly this area:

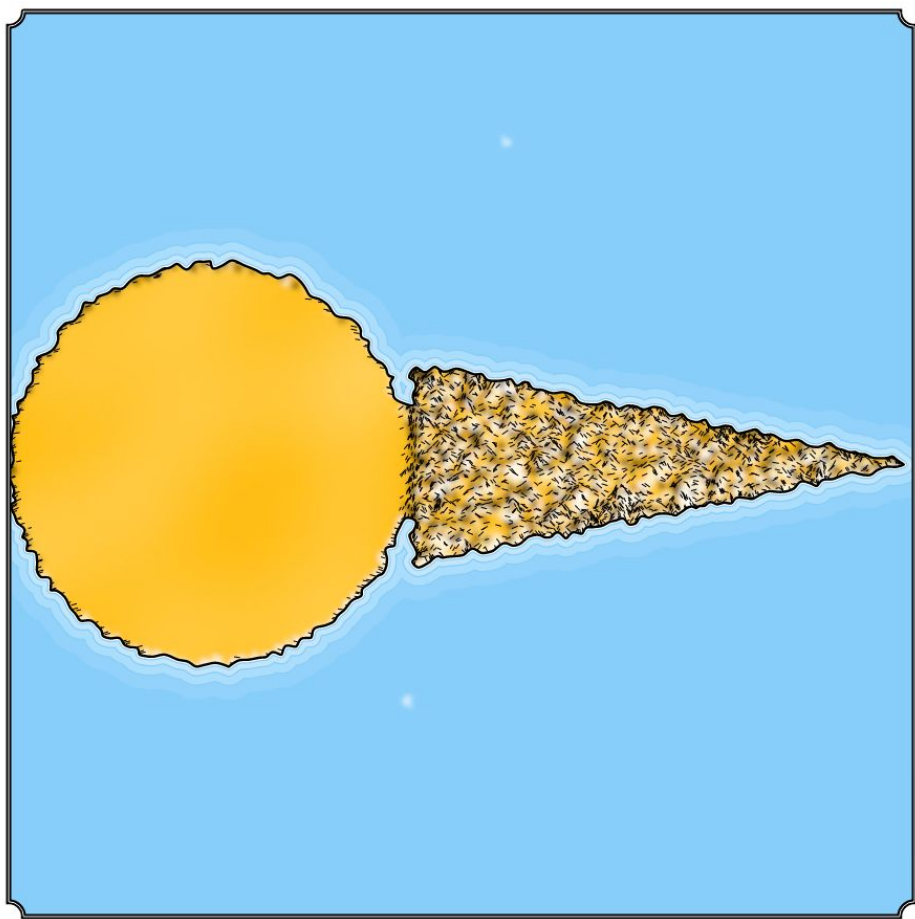


If the islands start dense and wide near the shore and get narrower and lower as they stretch out to sea, it will hopefully appear as if the same ridge that created the peninsula continues out to sea, sinking lower and becoming islands. So I can use the triangle shape as a mask and generate islands within that triangle. To test this out, I'll try filling the mask with a solid chunk of land. (I'm placing this off the edge of a simple circular land shape to get a notion of how it would sit in relation to a peninsula.)



That looks pretty good. Now I need to turn that flat land into a scatter of islands. How can I do that? Well, as the title of this article suggests, islands are just mountains up to their necks in the ocean.

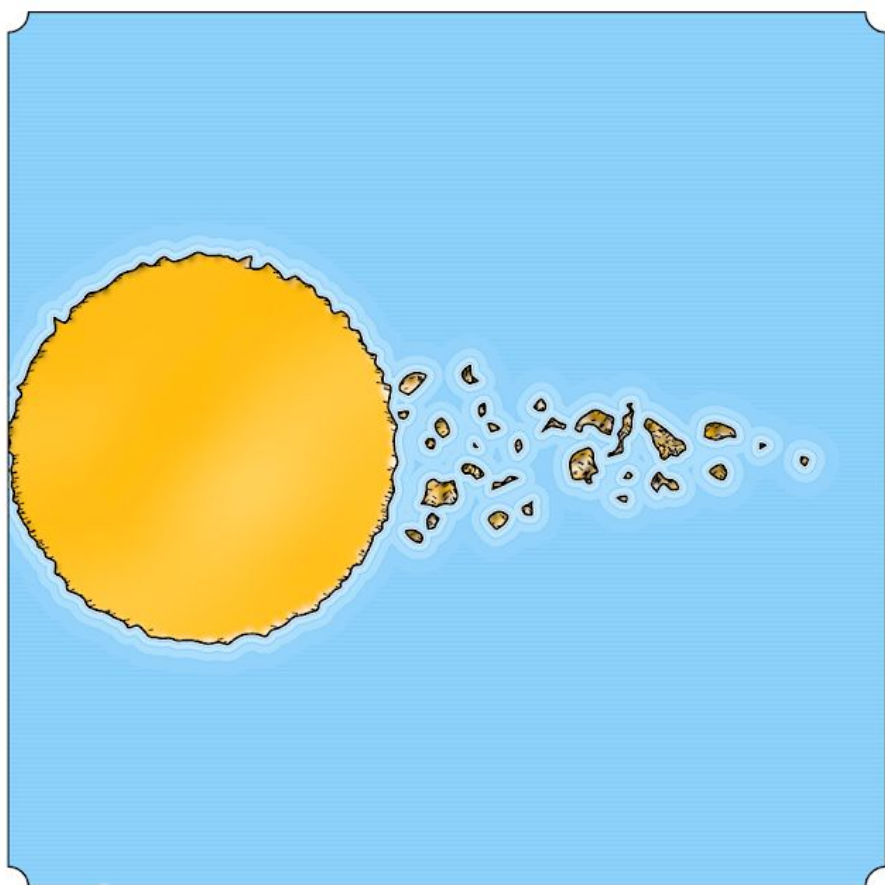
To start with, I will [generate mountains](#) in the mask rather than just flat land:



Here I've simply filled the triangle with terrain generated using Perlin noise. I used noise parameters that will give me a lot of small pointy mountains — three octaves of ridged multi-fractal noise. I chose this because I want to get a lot of small islands, but you can play around with different noise formulae to find a setting that you like.

Now I need to sink the mountains into the ocean so that only the tops are showing. With a little playing around with heights, I got this:





This already looks pretty good. I control how far to sink the mountains by setting a percentage of area within the mask that should be islands, and then I lower all the mountains until only that percentage is above water. (I'm showing more islands in these images than I think looks best on a map because I want the shape of the chain to be evident. In practice, I find about 25% land looks good.)

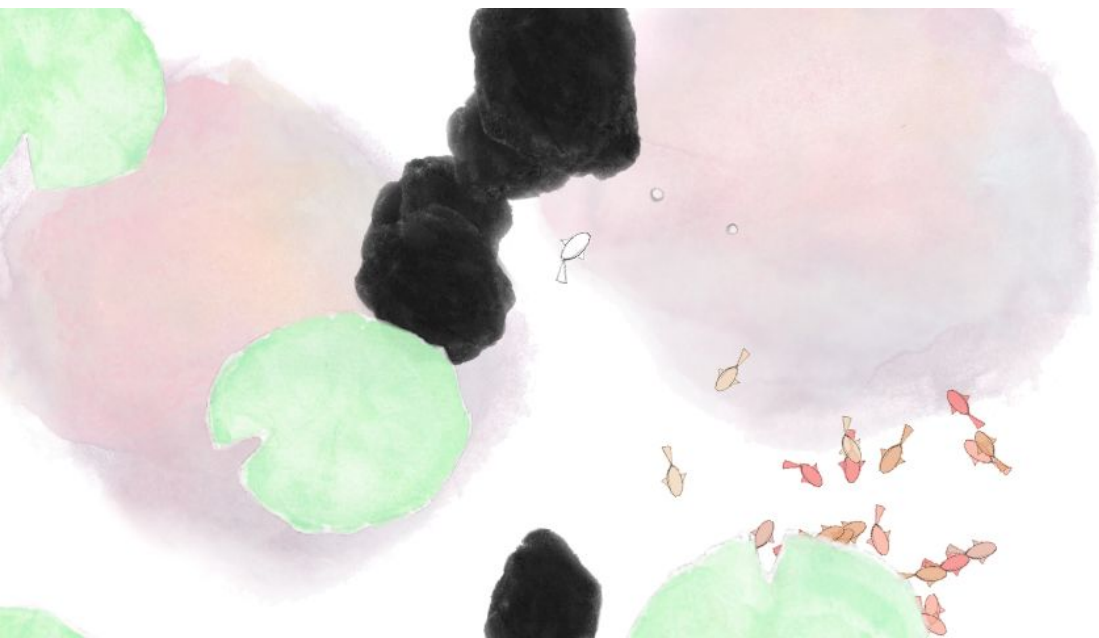
Learn more about generating islands in Part 2, on page 121!

Guppy - Progen as Antidote to Development Boredom

By Christiaan Moleman

@ninjadodo | <http://www.ninjadodo.net/guppy/>

I was working on an early prototype for my watercolor fish simulation, *Guppy*, when I started to get really bored with playing the same tiny test level with the same obstacles and hiding places again and again. I could have made a new level, or several, but it seemed this would only postpone the inevitable and I would soon grow tired of those levels as well...



Instead, I chose to randomize the location and number of lily pads and rocks in a level, effectively making the levels, such as they were, random. A primitive form of procedural generation to be sure, but it made the game fresh again and I could happily resume testing mechanics without the tedious feeling of rote memorization.

I gradually expanded on the generation: randomizing angle and scale (within a range), mirroring objects, creating multiple versions of art for smaller parts and randomly selecting between them for extra variety, rock formations that expanded into different shapes and directions, and level gen templates that specified particular combinations of rock and lily density and number of fish, etc.

I tied the complexity of generation to player progression so that early on (when you have a low hi-score) the levels are very simple so as not to overwhelm the player, but as you progress (with hi-scores of 5+ and 10+) the levels become larger and more varied. All this kept development and playtesting interesting and meant I was able to persist over the course of many evenings and weekends of spare time and ultimately ship my game.



Keeping motivation over the course of a longer project is hard enough as it is. Adding some dynamic variety can really help keep development enjoyable, making your game more replayable both for you and for players.




Game Randomizers: Procedural Variations on Your Nostalgia

By Jo Mazeika
@jomazeika

One of the *Hot New Trends* (trademark) on Twitch are game randomizers. Randomizers take a game, and shuffle around different parts of it, from the locations of items, to enemy stats and abilities, all the way up to generating entire new dungeons or overworlds. By doing so, they grant players the opportunity to explore a once-familiar space with new eyes. While they don't tend to alter the mechanics of a game (excepting some player conveniences or bug fixes), they instead offer a player the chance to explore a game in a new and unique way, recapturing some of the experience of being able to explore a world again for the first time. Many of these randomizers come out of speedrunning communities, and as such, most allow the players to specify the random seed used so that if players are racing to beat the newly randomized game, they're both on equal footing.

“Randomizers take a game, and shuffle around different parts of it, from the locations of items, to enemy stats and abilities, all the way up to generating entire new dungeons or overworlds.”

What makes the generation aspect interesting are the constraints that the pre-existing game imply for the randomizer. It's not enough randomly shuffle things; the game still needs to be beatable, and in most games, there are a number of configurations that will wind up unbeatable. To overcome this, constraints need to be placed on the system, leading to a complicated series of logic constraining different items and locations on the positions of others. Depending on the game, this can be a huge nightmare — the most popular randomizer, the *Link to the Past* randomizer (<https://alttpr.com/>), has around 30 different items that can be used to open up new locations within the game and all of these need to be accounted for when generating a new seed for players to explore. Other game randomizers, such as the *Dragon Warrior* randomizer (<https://github.com/mcgrew/dwrandomizer>) and the randomizer for the original *Zelda* (<https://sites.google.com/site/zeldarandomizer/>) also generate their worlds as well as shuffling the items — the *Zelda 1* randomizer shuffles the caves on the overworld, and has the ability to generate dungeons from scratch, while the *Dragon Warrior*



randomizer completely generates a new overworld, while leaving all of the dungeons and caves alone. In all of these cases, it's important that all of the important locations are reachable, and....

These systems represent an alternate approach to game generation — instead of trying to build a game generator from scratch, randomizers start from an existing game and explode its possibilities outwards, to the point where the generated experiences can be completely unlike the original's play. *The Link to the Past* randomizer has modes where finding certain items requires the use of glitches of varying degrees, allowing highly skilled players to test their skills. The *Final Fantasy 4* Free Enterprise randomizer (<http://ff4fe.com/>) turns the linear, narrative-driven game into an open-world treasure hunt by giving players access to two of the playable characters and an airship, and letting them explore the world from there. The *Final Fantasy 1* randomizer (<http://finalfantasyrandomizer.com>) not only gives players fine-tuned control over the gold and experience scaling within the game, but features overworld changes, designed by the community to make certain items more valuable as well as the ability to shuffle around all of the towns, dungeon floors and caves to make completely new spaces to explore. While most of these systems stay close to their original play experiences, each new feature allows for an even broader range of possibilities for these systems.

In short, game randomizers are a fascinating example of PCG out in the wild, generating new scenarios and new spaces for existing games. They offer a way of exploring the possibility space of a given game's world and mechanics, and they're a great way to re-experience a beloved classic in a new light.

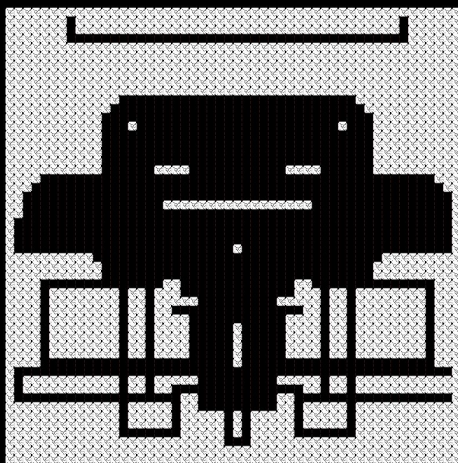
Designing Strata: A 2D Level Generator for Non-Programmers

By Matt Schell

@mattmirrorfish | <http://www.mirrorfishmedia.com>

In this piece I'd like to give some context on my thought process in creating Strata, my 2D level generation asset for Unity.

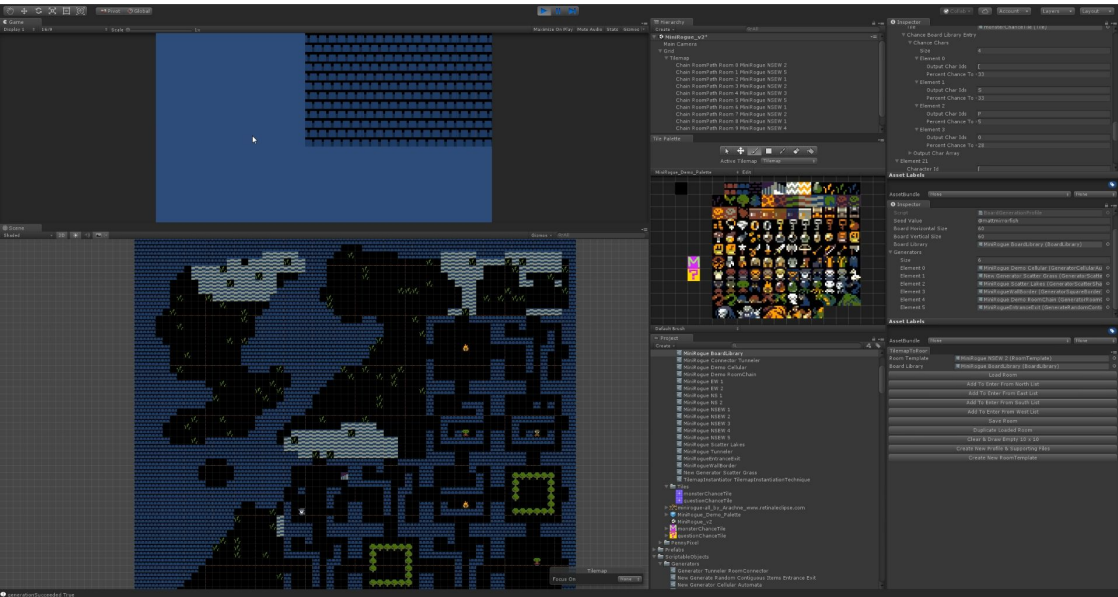
I was first exposed to the concept of procedural generation through the action-roguelike wave of games spawned by Derek Yu's *Spelunky*. I am a big fan of Jorge Luis Borges and his fantasies about infinite libraries, and procedural games struck me as an incredible way to play with the concepts of infinity that he imagined. Being new to game development (this was about 4 years ago), I immediately plunged into creating a 3D, flying procedural action game called *Monarch Black*. I had no idea what I was getting into and it's still not done. ͡(˘)͡




This summer I wanted a focused, finishable small project that I could actually ship that wasn't the albatross that *Monarch Black* has become in my mind. Hello side project!

My goals were that I wanted it to work with Unity's Tilemap tools, to be reasonably user friendly to non-programmers and to be able to create varied output without having to re-write the C# source code itself.

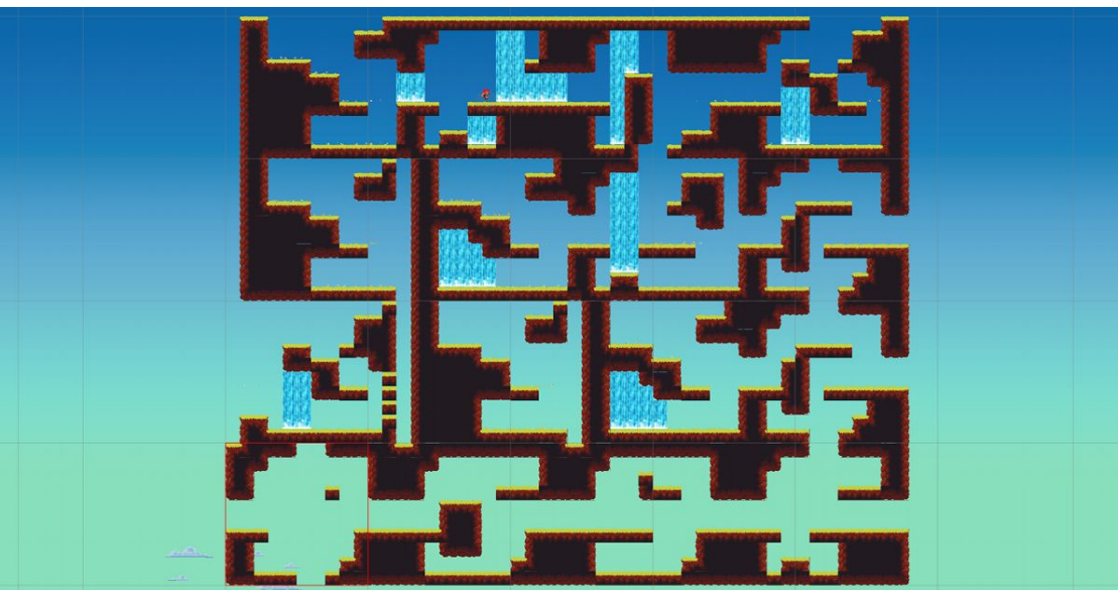
The use of Unity's Tilemap was inspired by reading about how Yu designed maps in Spelunky, with hand written blocks of ASCII characters representing level pieces that are then shuffled to create random levels. The mix of hand-made and procedural content is a nice way to address the often alluded to 'Thousand Bowls of Oatmeal' problem in PCG. Dropping some nuts and berries of authored content into the oatmeal can give it nice pops of flavor and texture!



Instead of using ASCII in Strata you can draw your level pieces using Unity's Tilemap tools. My hope is that this makes it more artist and level designer friendly. Under the hood Strata actually uses ASCII characters to build the initial version of the grid, before they're



turned into tiles to be displayed. Drawing tiles, then shuffling them to generate ASCII, then redrawing tiles opened up some options. I realized that since I have the tiles in this intermediate form, I could post-process and mess with them before turning them back into graphics. This led me to the current modular structure of Strata where you arrange what I call Generators into a series of generation operations on the same grid of data, overwriting and changing it as each one runs. By creating these Generators as custom ScriptableObject assets in Unity it becomes possible to choose, sequence and mix Generators via drag and drop in the Unity inspector.



This also allows people to choose between completely procedural levels, hand-authored levels or something in between. This is based on a software development pattern called Delegation. This means that non-programmers are able to control the generation process in an intuitive way. It also gives programmers who want to extend the system a clean point to begin doing so.



The more I started to work with Strata the more I realized that as much as it's a level generation tool, it's also a kind of drawing tool. I went to a high school for dropouts called City As School (which was amazing!) and I remember for our, very remedial, single required math class we were allowed to do a computer class. The class had very antiquated computer hardware and they had a 'turtle draw' style application which we were assigned to do things like 'input some commands to draw a square'. I remember figuring out that you could put in high numbers, randomize the colors and because it was so slow to draw you could generate long screen-saver-ish spirograph effects. Super fun!

Playing with Strata using the 'WanderTunneler' Generator (basically a random line/tunnel drawer) and the symmetry generators, I was having tons of fun making weird temple spaceship drawings. At some point I'd like to explore it more as a pure drawing tool, separate from game design. I've included a few pictures.

Because of the modular setup I intend to add a bunch of more sophisticated algorithms over time. I want to see what the initial response and level of interest is, and to get feedback from the community first, but I have a lot of ideas for other generation techniques to add.

It's worth noting that Strata is a paid tool, available via Itch.io and the Unity Asset Store.

If you'd like to learn more about Strata there is a page for it here!

Feel free to message me on Twitter via [@mattmirrorfish](https://twitter.com/mattmirrorfish) or to email me at matt@mirrorfishmedia.com if you want to chat!



Experiments in Generative Geometry: Building Meshes in Unity, Vertex by Vertex

By Alexander Pech


<https://rednax.itch.io/>

I'm an embedded software engineer from 9 to 5, but in my spare time I like to dabble with procedural geometry. Dissecting things into the smallest possible components and making them work for me is my jam, whether it's manipulating the registers of a CPU or finding new ways to coerce vertices and polygons into useful game assets.

Off and on in my spare time I've been jumping into Unity and, using the built-in Mesh class, just seeing what I could come up with. I started with the basics of programmatically placing vertices and triangles. Then I moved on to basic primitives like cubes, cylinders and spheres. Using these methods as building blocks I gradually progressed to more complex and organic shapes.

A particular area of focus for me has been trees. Trees lend themselves easily to this kind of algorithm for a number of reasons. The branching structure lends itself nicely to a recursive or iterative algorithm, similar to a fractal. The topology of a branch is fairly simple, really just a tapered cylinder. Also, all trees are unique but follow similar patterns, making them ideal candidates for creation via a set of rules plus randomness.

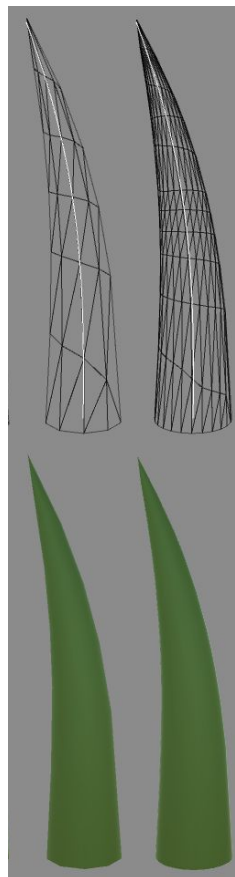




My basic procedural tree structure is a cylinder with some function controlling the direction of growth and radial thickness. I initially tried controlling branch direction via applying incremental rotations around a random axis. This gave a nice spiralled or twisted look to my branches but turned out to be quite difficult to control. My current solution uses spline curves. This gives me the control to generate a series of spline points however I want and then smoothly 'grow' the branch along the curve. I've also experimented with more physical based models, like growing towards a light source and applying gravity.

The radius function, even though it has less influence over the general shape of the tree, tends to be more complex. I like taking into account the angle of the current vertex as well to produce a bit of irregularity. This is particularly useful around the base of the trunk, where you don't always want a perfect cylinder. So, here you end up with what is essentially a two-variable function for your branch, defining the radius at any particular point along the length and radial direction.

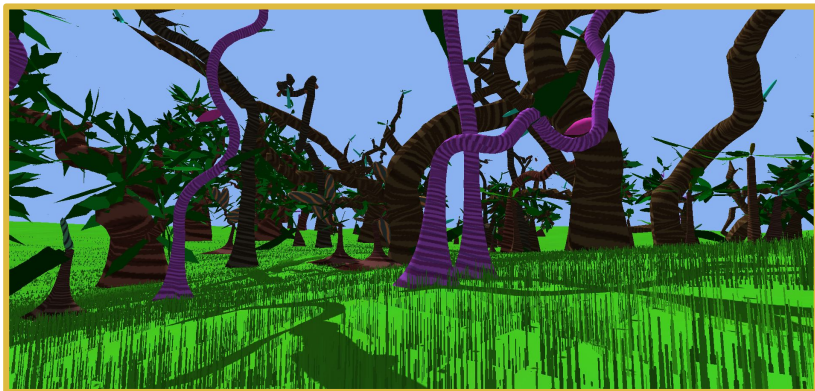
If this is confusing, here is the picture I have in my head when the geometry is being created. You start with a spline curve and at the start of the curve you draw an arrow perpendicularly out from the curve. The end of the arrow points to your first vertex. Then, you start to move the arrow up the curve while also rotating it, forming a spiral. The length of the arrow is controlled by our radius function. As it spirals up you periodically place a vertex at the tip of the arrow and connect to the surrounding ones. This is the basis for the current version of my algorithm. (You may have noticed the topology of these branches aren't exactly the same as cylinders as they have a single edge that spirals all the way along. Visually there is very little difference, but it gives me greater control because I don't need to place vertices ring by ring.)



From here I hope to keep building upon my toolkit, gradually producing more complex and interesting shapes and also making the meshes produced efficient and scalable so I can generate whole forests. This was really just an introduction into the methods I've used, as a lot of technical challenges have presented themselves along the way. Generating different levels of detail was a big one. At one point I threw out everything and started again just so I could have control over this aspect right from the start! But now I have a friendly 'polygon density' slider that does all this for me.

This has been a fairly technical explanation, but more than anything I'm inspired by the biological and evolutionary processes that created this life in the first place, so I hope to continue incorporating aspects of that into my models. Even though right now I'm just creating some funky geometry, there is a future vision that this may result in generating not just meshes, but functional environments with ecosystems and an evolutionary history. This will hopefully be just one piece of a vaster world-building engine.

For now, I hope my experiments find some value with others. You can find some of my work at my itch.io page <https://rednax.itch.io/> (including some random unrelated game jam projects). Hopefully plenty more to come!



From Hacker Poets to Cybernetic Poets

By Terry Trowbridge and Joseph Alexander Brown

@jb03hf | trowbridgeterry@gmail.com | jb03hf@gmail.com

Generative poetry and narrative programs have become mainstream projects for all levels of programmers and even gamer culture. Professional programmers design small scale poetry bots in their spare time. Undergraduate Computer Science students are frequently assigned little projects that test their skills with combinatorics and dictionaries. The ability to code a generative algorithm to mimic the vocabulary or style of a poet is a standard way to put the “STEM” in a “Science Technology Engineering Arts and Mathematics (STEAM)” curriculum. Literary criticism is quickly catching up, with more journals of literary studies asking for more generative poetry theory each year. Casual literary criticism in newspapers and blogs have moved away from asking “Is it art?” to more substantial reviews.

What is the next step for generative poetry? In a fast-paced world with short attention span, how does generative poetry resist a swift death as a fad, while also avoiding the slow stagnation of becoming more of a cliché than a genre? What justifies generative poetry as experimental instead of just reiterations of a standard toolkit? How is a poetry generator moving beyond just a developer of optimized literal strings of characters and into something of an art?

We propose that generative poetry needs to start working its way into live poetry readings, beyond the framework of production of poetry as an artifact and into the idea of poetry as performance. In particular, we believe that poetry programmers need to take their poems to the open mic sessions that bookend most community-based poetry readings. This aim is as much of a tactical maneuver, for recognition of the contribution they make to art in their local community, as it is a living laboratory to test their mettle like any human poet in the postmodern, intertextual, twenty first century.

“In a fast-paced world with short attention span, how does generative poetry resist a swift death as a fad...”

It is time for hacker poets to find out if they can “hack it,” pun



intended, in open mic poetry readings. Someone who codes generative poetry algorithms is contributing to the literature of their neighbourhood, city, and nation. They deserve some stage time among the other poets. We also suggest that while some poetry is cliché, all poets are eccentric and have a unique voice. All poets are cybernetic, in the sense of Wiener's definition of "the scientific study of control and communication in the animal and the machine." This creates a new definition of the Cybernetic poet who not only uses the tools but is able to present the expression.

Some read their poetry off of their smartphones. Other poets incorporate sound effects like looping and distortion in their readings. Others add images to their stagecraft that doesn't appear on the page or poetry blog. Some poets have artificial limbs, speak through vocoders, or simply change the settings on the microphone/amp setup on stage. All manner of performance poets utilizes the live venue to create unique artistic experiences.

We assert that generative poetry is equally as cybernetic as the other forms of poetry that happen on stage, especially when computer generated poems are performed through the voice, body, and presence of a human. We will not know what that means for each AI poet, however, until they take the stage. The Turing test is not enough to examine a poem and only looks at the ideas development method, and not the importance of the expression of the ideas on stage.



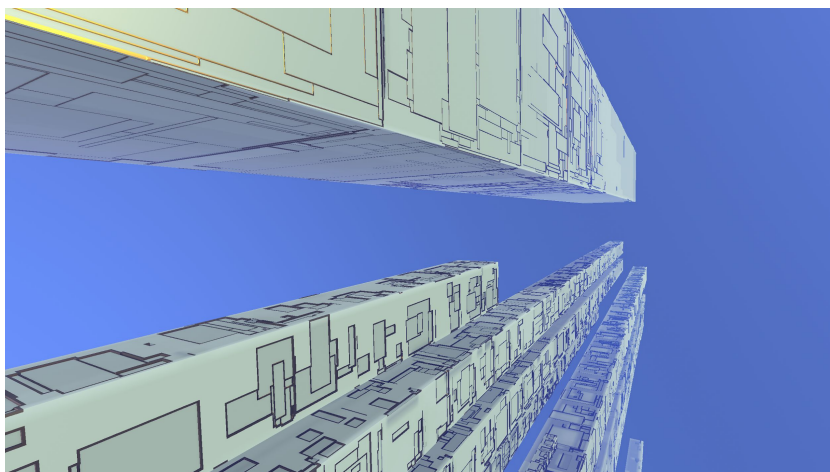
Boxes in Space

By Juliette Foucaut and Doug Binks

@juulcat | @doughbinks | <https://www.enkisoftware.com>

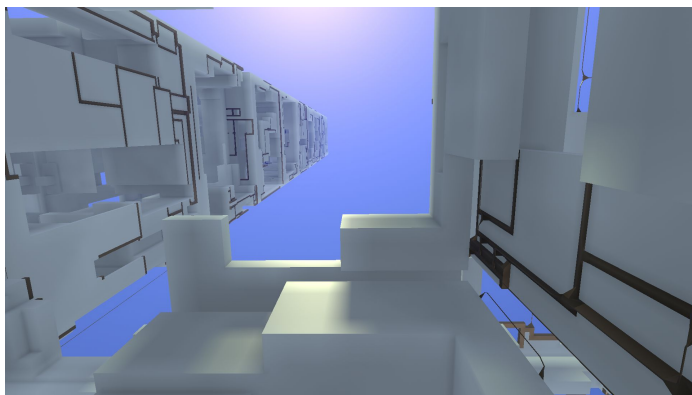
We use procedural generation to create a 3D environment for the prototype of our game, *Avoyd*. Our main reason for employing the technique is to keep the download size small. *Avoyd*'s a voxel game in zero gravity. This means that all surfaces can be traversed in any direction: players can fly but also attach themselves to any surface and hover. In addition volumes can be edited in-game. The game prototype features a fair amount of destruction, so we mix materials of varying hardnesses to show off these properties.

Inspired by concept art by Rebecca Michalak (<https://www.enkisoftware.com/devlogpost-20160527-1-Concept-art-by-Rebecca-Michalak>), we went through many iterations and by trial and error ended up with a structure made of clusters of boxes linked by bridges. Using voxels simplifies much of the procedural generation since operations such as adding and removing volumes are fairly trivial to implement. Most of the variables such as box dimensions and group sizes are randomly chosen within boundaries. Here's how we progressed from a simple voxel cube to clusters of greebled boxes made of a selection of materials:

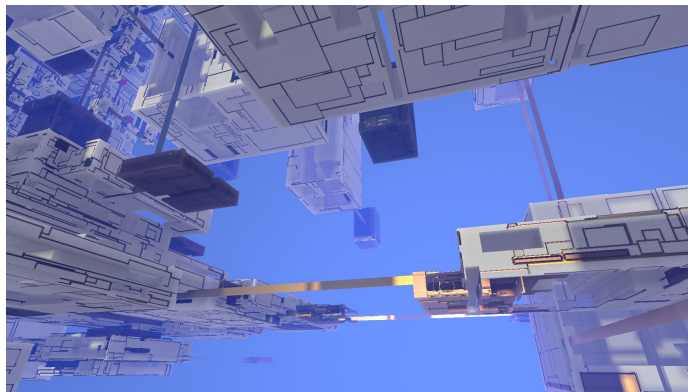



To get these towers, draw a white cube, add greeble to the whole surface by adding a random number of flat metal rectangles smaller than the side they're added to, and hollowing them out by removing a box 2 voxels smaller offset by 1 voxel to the center of the metal rectangle.

Move one cube length along an axis, add a new cube and greeble it, repeat.

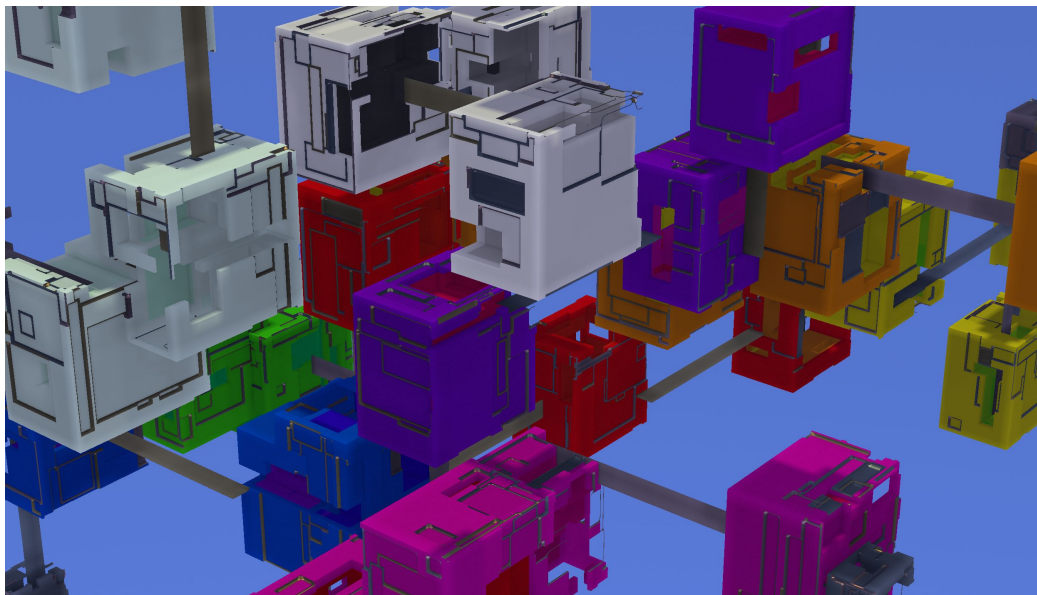


To hollow out a cube, delete random smaller boxes inside.

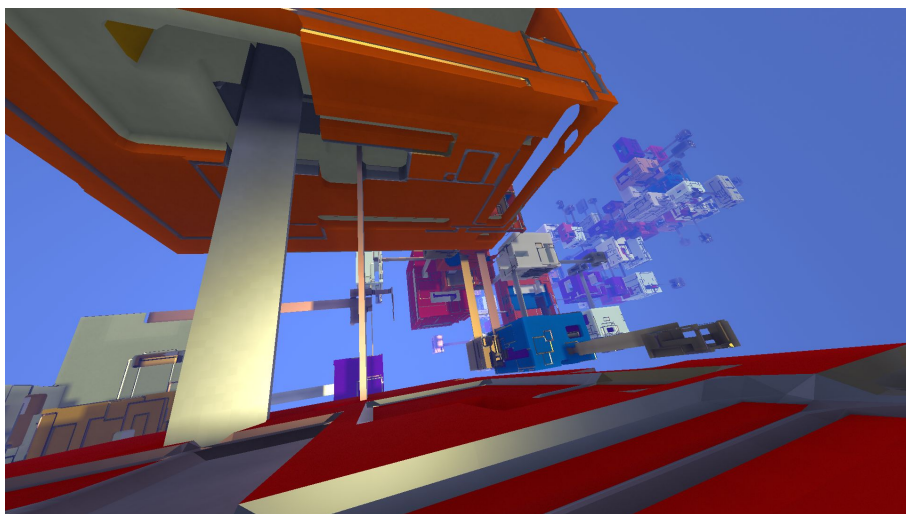




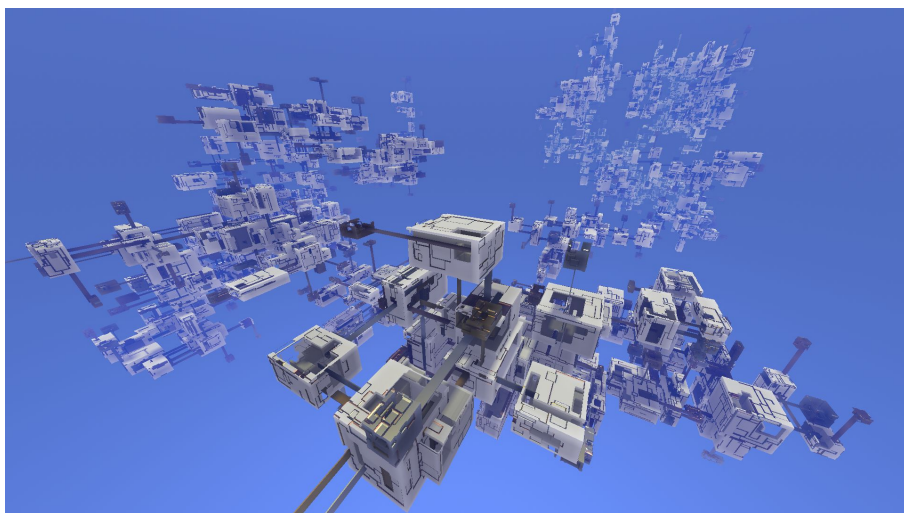
Cubes become boxes (random dimensions along each axis). Some of the surface greeble is not hollowed out: they look like metal plates. Instead of stacking the boxes along an axis, they are placed randomly side by side and linked by planks/bridges. From one side of each bridge we create a navigable path by removing a box shape the width and length of the plank bridge but taller. Some of the metal planks are dead ends that lead to a metal "safe room" (metal is a harder material to destroy) built using the same rules as the white boxes.



Randomly colour the boxes using neighbouring materials (materials are identified by incremental numbers and colours). Each box is first created full, then we overwrite the inside with smaller boxes made of different materials/colours. Following that, each box is greebled and hollowed.

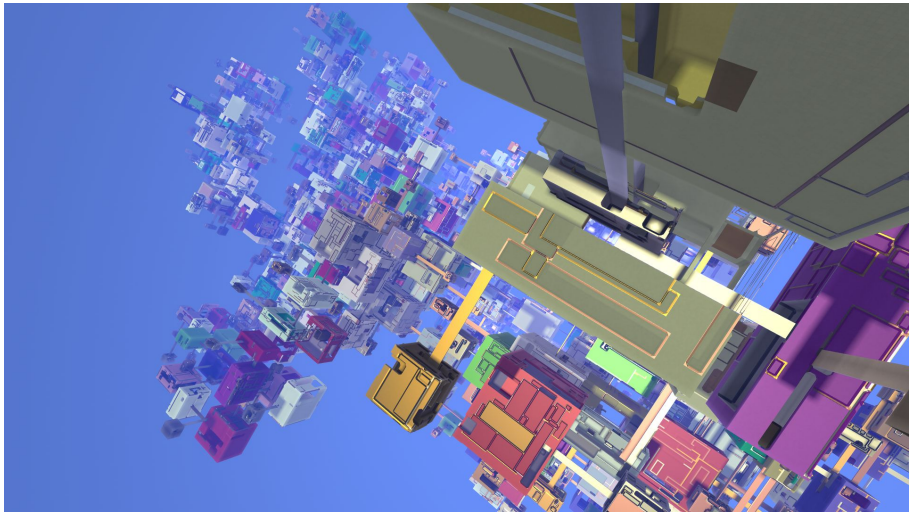


Here we're only using one materials family (colours white and grey) to take a closer look at grouping the boxes in clusters.

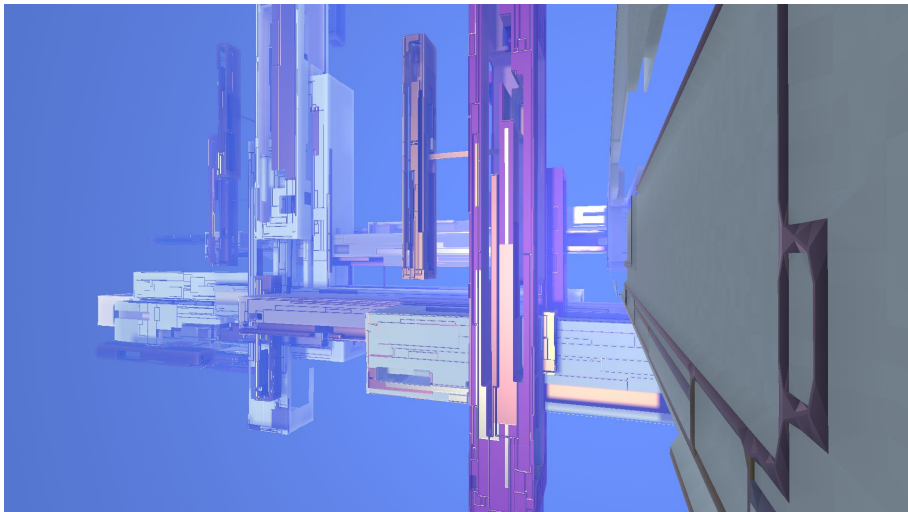




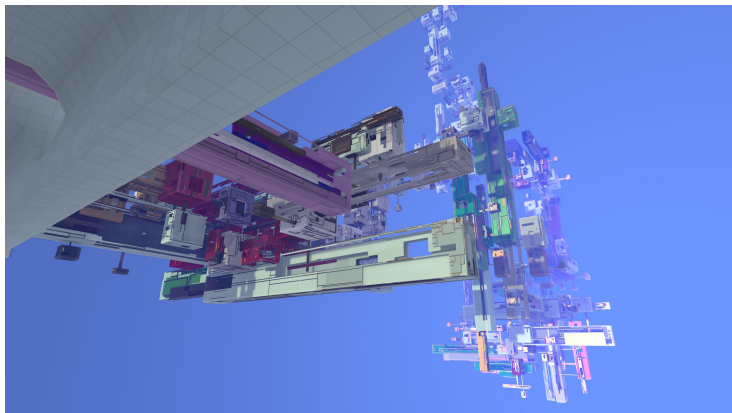
Grouping in clusters and multiple material families (colours) combined



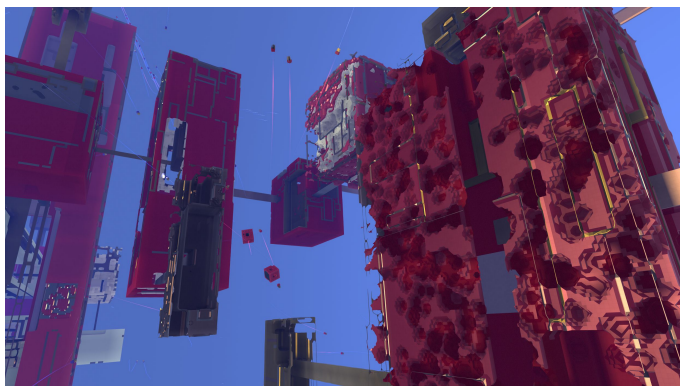
As before, but in a very large world. It looks a bit messy.




Trying to add structure and landmarks by lengthening some of the boxes a lot.



Grouping into clusters again: each cluster has a dominant axis the boxes are lengthened along, and a single material family. Each material family has a dominant colour: white, ochre, red, green, blue, and various strengths. The white family is used about twice as often as the others, though this is only detectable in large worlds made of many clusters.





During gameplay, players and drones firing damages the generated world. This is damage caused by a drone battle in a red cluster. Strong materials take less damage than weak ones. In the current build, Avoyd 0.3, materials do not shield each other.

To get an idea of the result, you can download Avoyd from our website

[\[https://www.enkisoftware.com/avoyd\]](https://www.enkisoftware.com/avoyd).

When playing the game, the procedurally generated boxes described above are the default environment. The game also features a Voxel Editor which can be used for generating the procedural boxes. A small number of parameters can be tweaked. To familiarise yourself with the editor, follow the tutorial in Avoyd's Voxel Editor's "Help > Tutorial" to "Create a simple shape with 'Set'". To try out the procedural generation, 'Set' the shape called "Procgen: Linked Boxes". Use a large box size for better results although the scene will take longer to generate.

Carving the Infinite Plane into Discrete Regions

By Keith Evans

@kaynSD

Unlike *Infamous* or *Daggerfall* which utilised procedurally generated content (PGC) during development to build staggering amounts of content that would later be refined by traditional designers, games like *Dwarf Fortress*, *RimWorld* and *Ultima Ratio Regum* generate vast, persistent spaces for exploration at player request, each world generated from a single random seed at the start of a game.

A subset of these kinds of spaces are those that exist on an infinite plane. Rather than limit themselves to a fixed area, games like *Minecraft* potentially go on forever in every available explorable direction. Traditionally built by layering many Perlin or Simplex Noise algorithms of different scales atop each one another, the generator will continually feed a player an infinite horizon by observing the values returned from inputting a given pair of cartesian coordinates to the noise functions, and interpreting the returned values as new tiles (in *Minecraft*'s case blocks of tiles are built at the same time and stored as chunks).

There are limitations to using noise like this; chief amongst them is it can often appear nonsensical. Should a player take a bird's-eye view of the world they will see a scattered, abstract pattern with no clear oceans, continents or identifiable features.

From a designer's point of view it is also difficult to design a game's mechanics that operate on properties of a map that are wholly random in nature. How often a specific feature of a landscape is generated can be at the mercy of luck, making for interesting terrain but can mean unpredictable gameplay length and pacing.

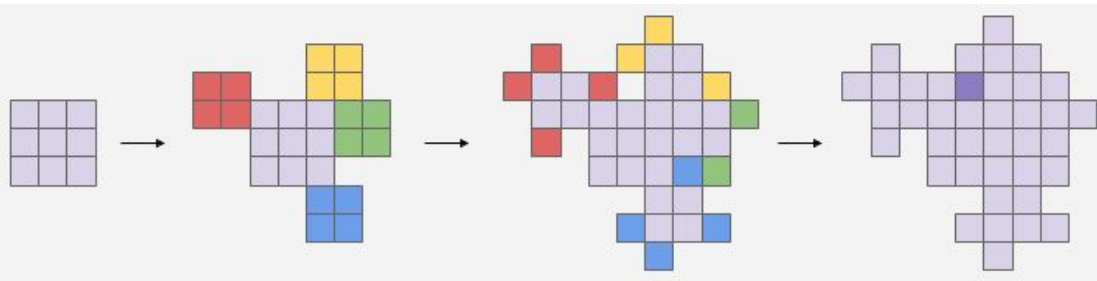
To address these issues, I have been working on an algorithm to carve an infinite plane into discrete elements. Rather than refer to multiple noise functions to develop individual tiles and chunks in isolation, my generator creates fractal regions on request at runtime.

These regions are collections of chunks connected together and generated simultaneously; which know of each other during their generation steps and can build content in relation to each other and in the context of neighbouring regions.

These regions can be thought akin to Zones in an MMORPG such as *World of Warcraft* or *Final Fantasy XIV*; they are bounded discrete areas of a given size that know which other zones they are connected to. By carving sections of the infinite space into discrete regions, the problems inherent to infinite space are reduced but still allow for endless exploration.

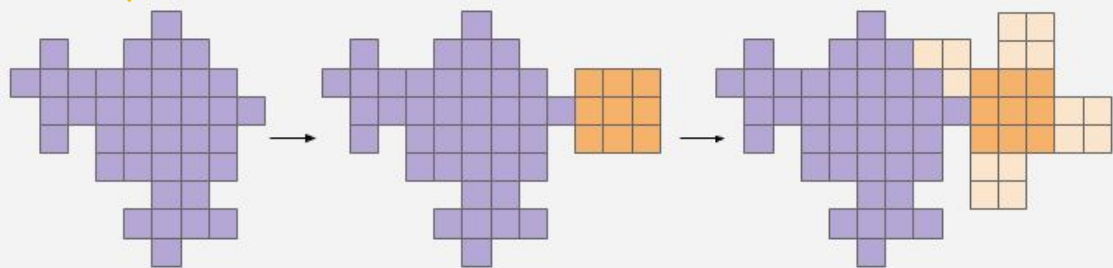
The initial pass of the algorithm defines the shape of the area, generated in a variant of a Koch Snowflake. Starting with a large blocked out square of chunks, from a random point on each of the four edges, a smaller square of chunks is extruded outwards. This repeats recursively down to a single chunk (figure 1).

Figure 1



Since these areas are extruded outwards from a given point, rather than simply placed, the algorithm can account for already generated chunks, and can be cancelled at any point along its extrusion, generating areas that are wholly contiguous. This area becomes the base region (figure 2).

Figure 2




In the second pass, the generator runs a flood fill sweep from just outside the determined boundaries of this base region (and any of its detected neighbour regions), to find any potential chunks that have been orphaned by the process, i.e. those that are wholly enveloped by the base region and its neighbours (**Figure 1**). These orphaned chunks are added and this forms the region.

Once the physical dimensions of the region and its currently generated neighbours of are determined, other steps can proceed. These may include:

- Climate Assignment, which can be influenced by neighbouring regions to prevent tropical desert zones emerging next to stretches of arctic tundra.
- Constructing impassable mountains, coastlines and passes at the perimeters to control travel between the region and neighbours
- Smart placement of gameplay elements; settlements, rivers, roadways and easily identifiable landmarks (“weenies”) can be sensibly assigned to this discrete space, allowing the region to have its own identity.

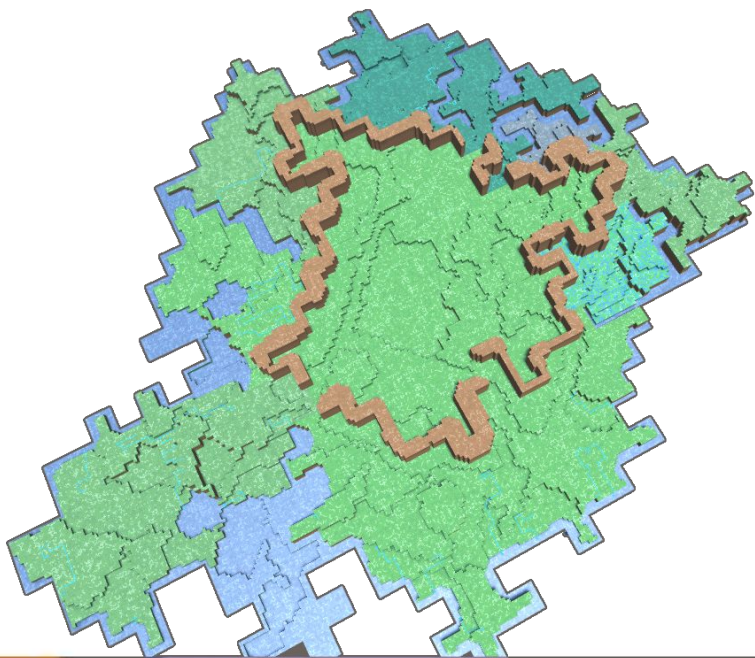
A region can be created at any point in the infinite plane, with the knowledge that they will connect seamlessly to other parts of the map once new adjacent regions have been generated. This allows for



the placement of fixed, future or mandatory content, allowing control of the game's pacing.

As with all PGC, there are trade-offs in algorithms selected. In this case the order that the regions are generated in changes and influences how regions will be generated in the future, influencing their geography, climate and contents. This makes it impossible to predict what properties any given hypothetical tile will have until the region containing it starts generation, unlike with deterministic noise functions which will always be able to identify what a tile's content will be simply by passing the algorithm a pair of X,Y coordinates

Despite this complication, this region focused method of generation has potential, and could find a place in any number of games wishing to boast infinitely explorable worlds whilst allowing for context sensitive positioning of gameplay elements.

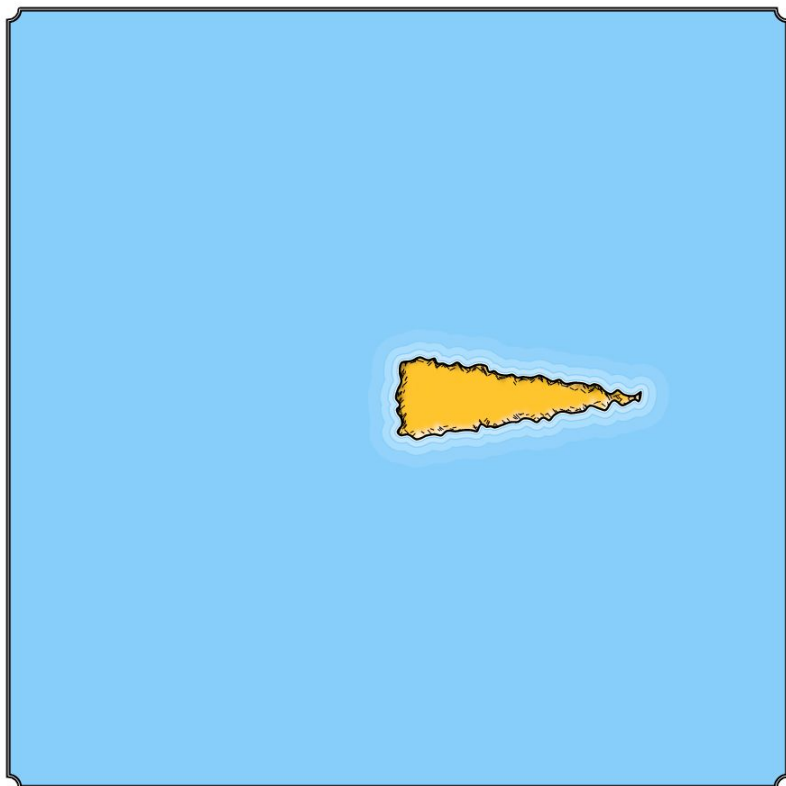


Islands Are Just Mountains Up To Their Necks in Ocean: Part 2

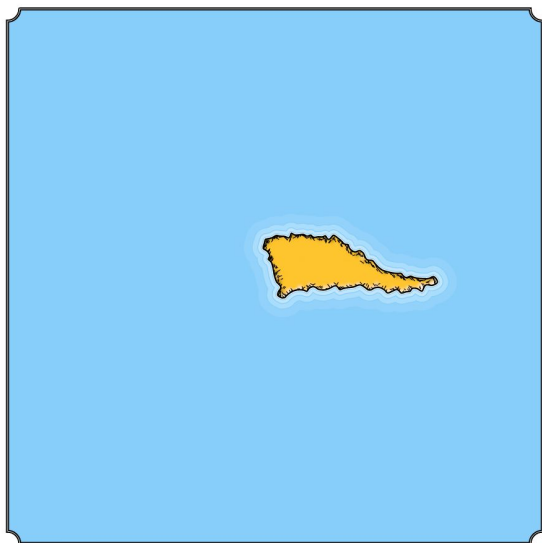
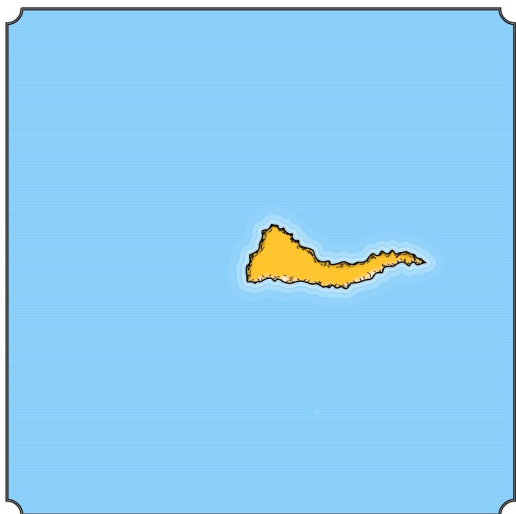
By Scott Turner

This is Part 2 of Scott's island generation article! For Part 1, flip back to page 90!

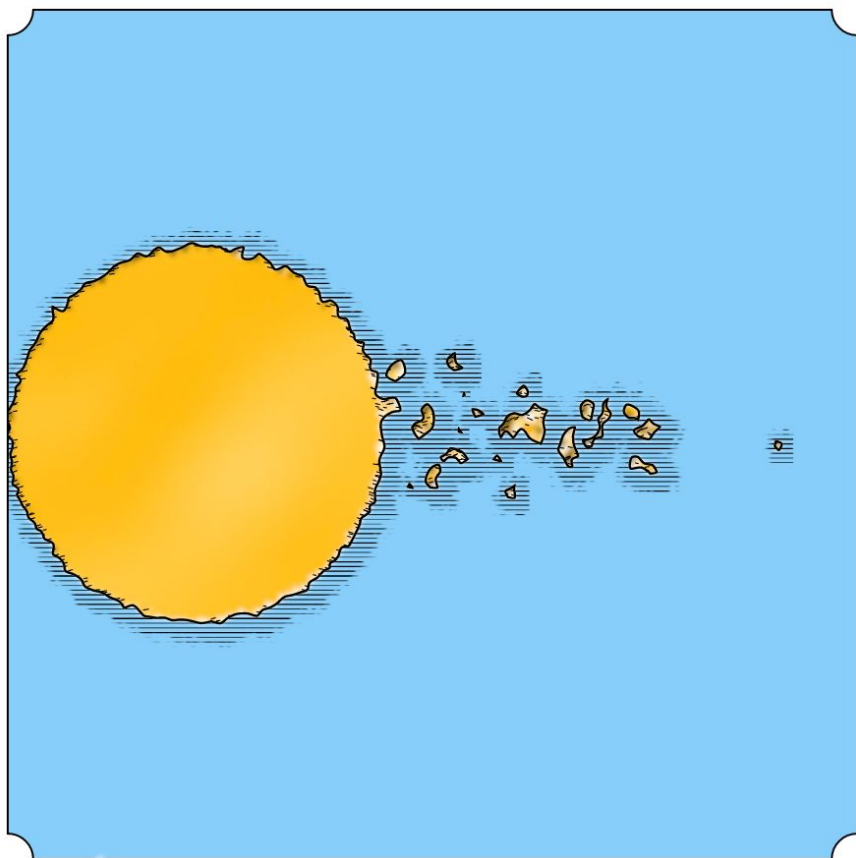
For a further refinement, I can perturb the shape of the mask with noise to get something that is less regular. You can read in more detail about [using noise to perturb a basic shape](#) on my blog, but the basic idea is to move the edges of the shape based upon noise. The noise for two points that are close to each other will be similar (that's what distinguishes noise from random numbers) so the edges will be distorted (or perturbed) in a consistent way. To see what this looks like, I'll start with the basic triangular mask:



And here are some perturbed versions:



I'm using fairly minor perturbations here. It's important that the island chain point away from peninsula to be convincing, so I don't want to use so much perturbation that I lose that basic arrow shape. Here is an example of an island chain created with a perturbed mask:



I'm using fairly minor perturbations here. It's important that the island chain point away from peninsula to be convincing, so I don't want to use so much perturbation that I lose that basic arrow shape. Here is an example of an island chain created with a perturbed mask:

Perturbing the mask has made the arrowhead shape less obvious and the overall shape of the peninsula islands more natural and organic.

Now that I can create an island chain, the next step is to identify candidate peninsulas on the map and figure out how to properly align the mask with the peninsula. Sadly, the research field of peninsula identification seems to be moribund, so I was forced to invent my own algorithm.

A peninsula is a point on the coast line where the land bulges out into the sea. To detect these spots, I slide 2 points (Beginning and End) along the coast line, keeping them a fixed distance along the coast apart. At each step, I check whether the midpoint of the line between Beginning and End is over land. For example:



Here the green point is on the midline between Beginning and End and is over land. So I know that the coast between Beginning and End forms a peninsula. If there is water, I have a bay:



(In practice, it's better to check a number of points along the line between B and E and make sure they're all land, because in some situations where B and E are straddling a long narrow bay the midpoint might be land with water to one side.)

Once a peninsula is detected, the next step is to determine where to anchor an island chain and which direction it should point. This starts by placing a new point on the Middle point of the coastline between Beginning and End:

This point is the anchor for the island chain. To measure how “pointy” the peninsula is, I create a line from the Middle point to the line between Beginning and End:



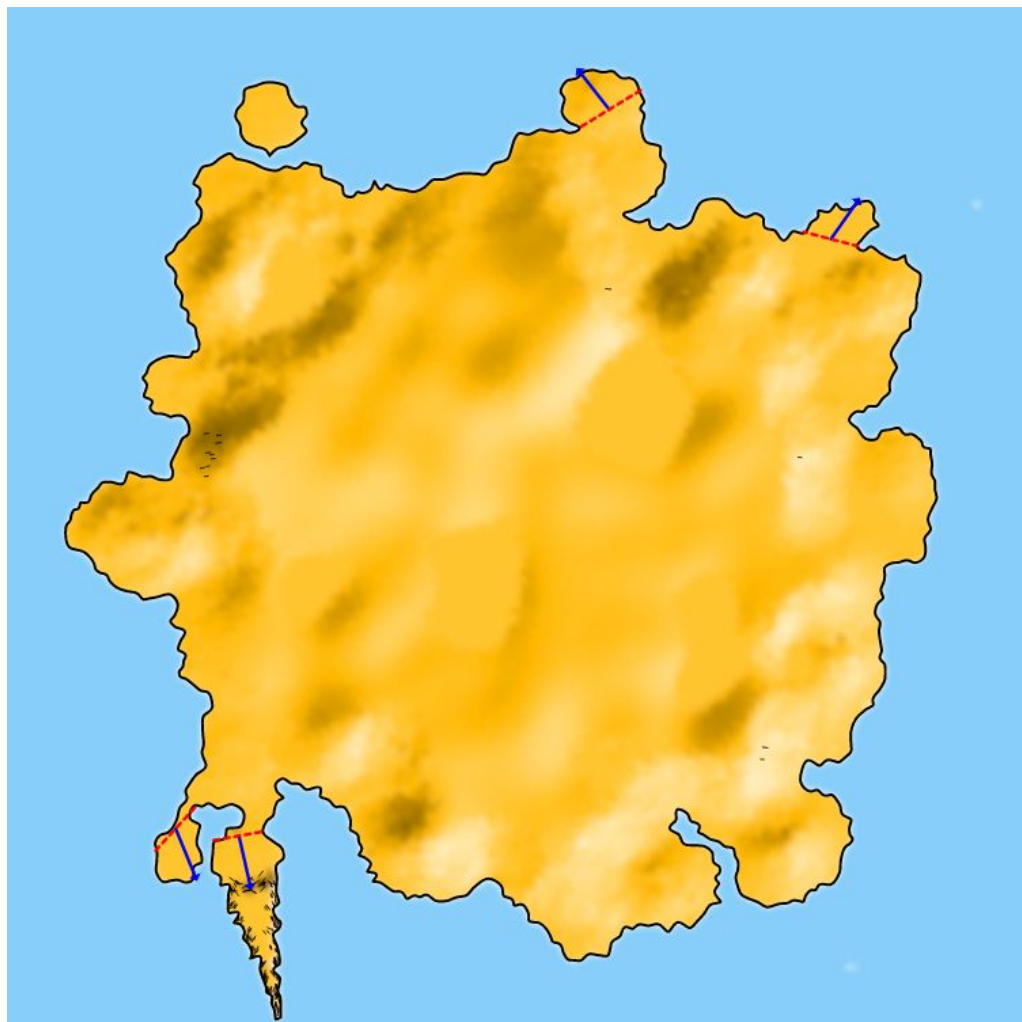
The ratio of the length of this line (the blue arrow) to the length of the line between Beginning and End is a measure of the peninsula's “pointiness”. Pointier peninsulas are better for anchoring island chains. I don't want to put an island chain off every peninsula, so I scan the coastline, rank candidate locations based upon their pointiness, and then use the top candidates.

Here's an example island (generated with Perlin noise) with the top four peninsulas highlighted:



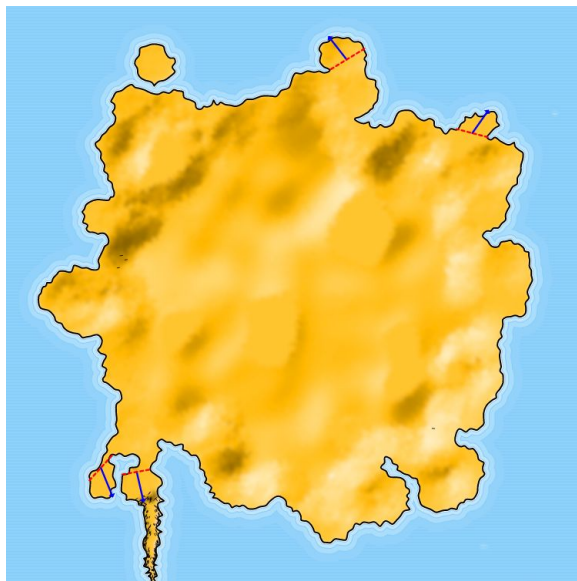
You might note that the blue arrow isn't always perpendicular to the red dotted line. This happens because "M" is at the midpoint of the coast line between "B" and "E". Since the coastline is wiggly, the midpoint is not always exactly in the "middle". In the example at the lower right part of the map you can see how sliding the red dotted line along the coast sometimes finds somewhat odd "peninsulas". However, in general the algorithm works pretty well.

The best candidate on this map turns out to be the lower middle peninsula, so that is where I'll add an island chain. To size the mask for this chain, I'll make the base of the wedge the length of the dotted red line, and I'll make the length of wedge about three times the length of the blue line. On the following map I've filled in the mask with solid land to illustrate it's size and shape:

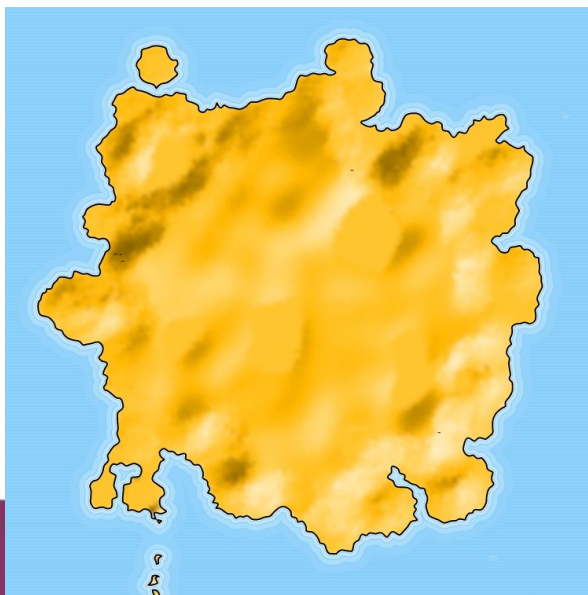


Keeping the mask no wider than the peninsula helps it look like a continuation of the peninsula. The length is more a matter of taste — choose what looks good to you.

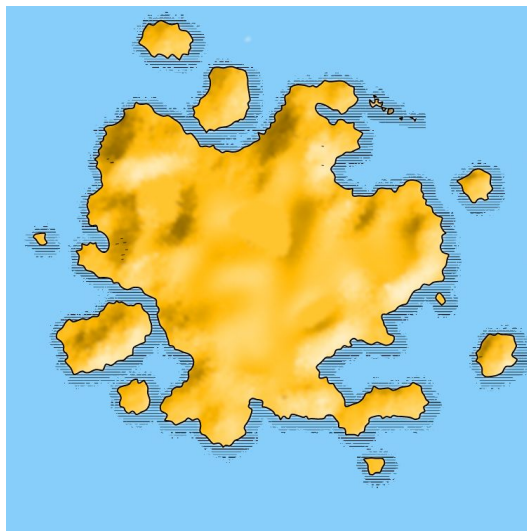
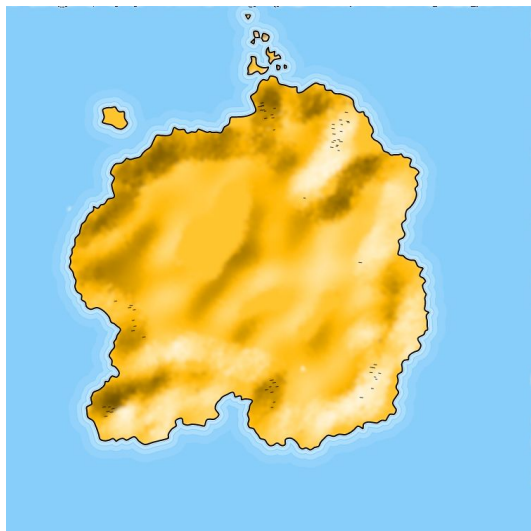
Before creating the island range, I perturb the mask to make the regular shape less obvious:



And now I can generate the islands:



Here are a couple more examples:



That covers the basic process *DRAGONS ABOUND* uses to create small archipelagos. As I hope the explanation makes clear, this process uses noise (both to perturb the basic mask shape and to create the mountains) but harnesses the noise within a procedural generation algorithm that creates a specific kind of terrain. This enables *DRAGONS ABOUND* to create novel maps that also suit a particular purpose. For example, if I wanted to generate a map for a fantasy role-playing campaign that started with the players being shipwrecked, I could use the archipelago island generation to make a map which I could be certain would have some suitable rocks for a shipwreck. The combination of a structured procedural generation algorithm driven by noise can be a powerful method in your developer's toolkit.

Apop: Seeing Patterns Everywhere

By Isaac Schankler

@piesaac | <http://isaacschankler.com>

At first I didn't find cellular automata very interesting. As a simple way to generate complex patterns, yes, they were neat, but the black-and-white pixelated pyramids they tended to generate had a samey quality that didn't seem super artistically interesting to me. It wasn't until I saw Gwen Fisher's "Pixel Paintings" that I changed my mind. Gwen, a mathematician and visual artist, had incorporated colors and imperfections into her use of multistate cellular automata. The paintings hovered somewhere precariously between abstract and representational art: you could see, or imagine, lava flows, rivers, hanging gardens.

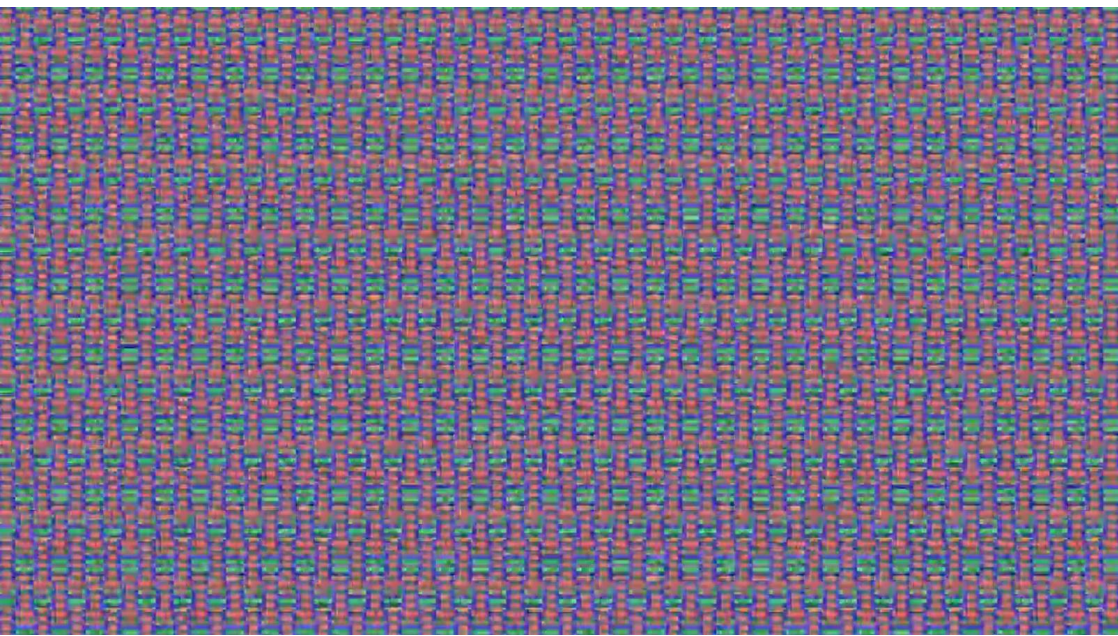
"The results were immediately compelling, and fun to play with.."

Immediately I thought this had musical implications, and I set about mapping colors to everything I could think of: pitches, samples, timbres, more esoteric parameters. The results were immediately compelling, and fun to play with. For example, the initial conditions (or seed) of an automaton can have a great impact on the automaton's evolution. Seeding a particular automaton with random values causes it to evolve into other random-seeding patterns (Figure 1), while seeding it with a row of a single value causes it to settle into a periodic fluctuation of more rows containing only a single value (Figure 2). Initializing it with the seed 111121111112111 yields a more intricate and coherent pattern that displays many symmetries and near-symmetries (Figure 3). This seed can be thought of in musical terms as a backbeat, with accents on beats 2 and 4 of a 4/4 measure. In other words, seeding an automaton with a musically cogent patterns can yield other musically cogent patterns. Another interesting feature of this pattern is that it is a long-term stable structure; after 41 rows (or measures), it repeats.

The patterns also respond satisfyingly to interventions, like putting your hand underneath a faucet and watching it change the flow of the water below. They also present an interesting challenge to the listener/observer, who has to decide what information is most



musically relevant. This perception can change depending on how the automata are turned into sound, from noise to percussion to melody and harmony. These shifting representations seem to interrogate how we as listeners derive musical meaning from patterns, and in larger sense, how we decide what information is relevant to us. Often in looking for patterns we see something that's not there. It turns out there is a lovely word for this: apophenia. The abbreviated form of this, Apop, became the piece's title.

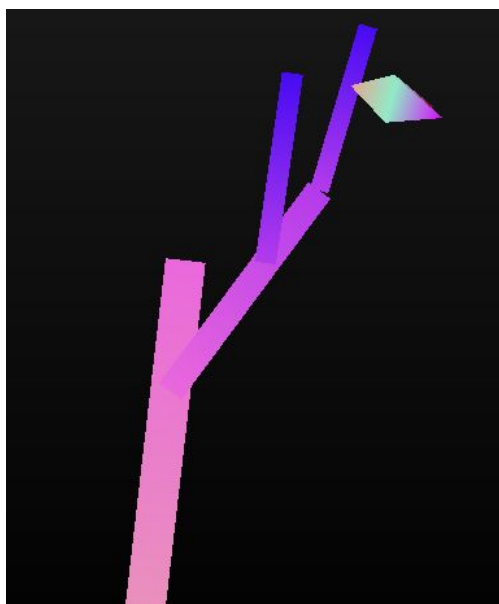
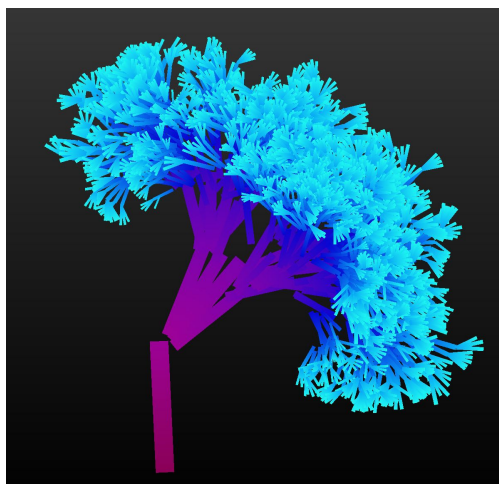


Alien Growth - A 2D Plant Generator

By Tilman Schmidt

@Keymaster_ | trowbridgeterry@gmail.com





But What Do You *Do*: Mechanical Ideas for Turning a Cool Generator Into a Compelling Game

By Prophet Goddess


@prophet_goddess | <https://prophetgoddess.itch.io>

An experience that I — and maybe you, given the demographic of this zine — have had often goes as follows. I have an idea for a generator. “I’d like to make a planet generator”, or “Let’s make something that generates islands”, for example. Then I spend way too much time working out an algorithm to generate this thing I want. Then, upon exploring my new creation, I realize I have no idea what this generator is for besides being kind of neat.

Now, generators that are just kind of neat are fine! This isn’t a manifesto against games/non-games where you just explore a space created by an algorithm. Yet, I like to aspire to more. A good generator becomes so much more appealing when paired with mechanics that encourage the player to interact in ways that show what’s most interesting about the space. This isn’t just true of procedural games: all games enjoy design decisions that force the player to see what’s cool about them. With that in mind, here are a few mechanical ideas for encouraging players to explore your latest generator:

#1: Photography

To explain this let’s talk about a game that isn’t procedurally generated: *Dead Rising*. *Dead Rising* is a 2006 action-RPG about zombies who overrun a mall. You play as photojournalist Frank West, who has covered wars, you know. Frank’s occupation leads to one of the game’s oft-overlooked mechanics: Frank can take pictures with his camera, which awards the player with experience points based on how well they fit certain criteria. The fact that nobody has, to my knowledge, made a procedurally generated game based around this concept baffles me. *No Man’s Sky* comes close, rewarding you with money for scanning new plants and animals you discover, but that’s far from the core of the experience. It also doesn’t encourage the player to engage with the space in quite the




same way as an actual photography mechanic. Photography mechanics can give players guidance while still letting them set and complete their own goals.

#2: Delivery

Let's talk about another game that isn't procedurally generated: *American Truck Simulator*. For the unfamiliar, *American Truck Simulator* is exactly what it says on the tin — a game where you play as a truck driver making deliveries in America. It's a sequel to the also much-acclaimed *Euro Truck Simulator 2*, which is also exactly what you think it is. The Truck Simulator games are a perfect example of how very minimal structure can transform an experience. In *American Truck Simulator*, you get loads to carry from point A to point B, and that's about it. The main draw of the games is experiencing the lovely sights along the way. A procedural version of this idea is fairly easy to imagine, but it doesn't need to just be a game about road trips. One could also imagine a game where you fly your spaceship between planets making deliveries. Or you could be sailing a boat between islands. Or walking on trails through the woods. The beauty of this mechanic is its simplicity. It's easy to implement to work with your specific generator concept compared to the trickier task of systemizing what constitutes a good photograph. Like photography, though, it doesn't bog the player down with systems to draw their attention away from the real star of the show. It also provides opportunities for procedural narrative and character.

#3: Archaeology

Have you played *Caves of Qud*? Again, given the demographic of this zine, I'm willing to bet you're at least somewhat familiar with it. There's a lot to love about *Caves of Qud*, but one of the most compelling aspects of it is its procedural archaeology. *Qud* is, in large part, a game about exploring the ruins of ancient civilizations,



piecing together the past through the fragments left in the present. If you're committed to your experience being without such extrinsic motivations as points and quests, archaeology provides some light structure that allows maximal freedom for players to choose how to engage with your game. It is, however, much more involved to implement a consistent and compelling procedural history than it is to implement photography or deliveries.

This is not a conclusive list of non-violent mechanics for encouraging player exploration. I encourage you to think about these as starting points. You can borrow them wholesale for your game, or you can use them to spark inspiration for similar mechanics all your own. What matters is taking advantage of the power of game design to show what makes your generator special.

High End Procedural Systems (Procedural Trip Report)

By Denis Kozlov

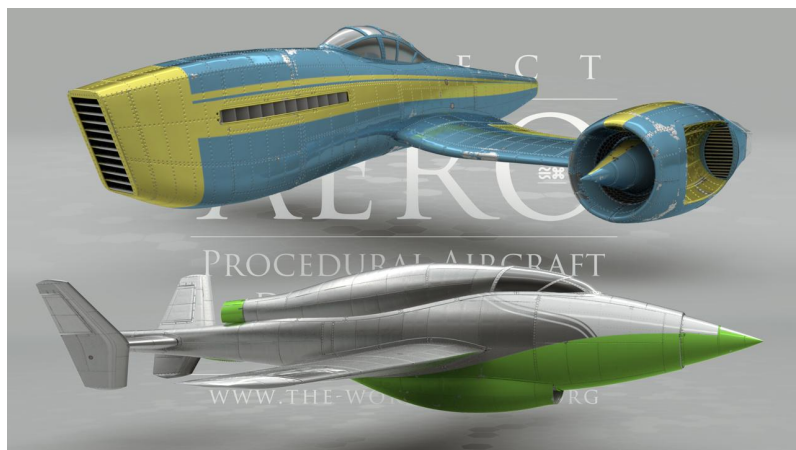
@kozlove | <http://www.kozlove.net> | <http://www.vimeo.com/kozlove>

My background is in commercial CGI — Mostly film VFX and advertisement; gamedev, television and even a touch of DTP too. I've been attracted to proceduralism for most of the career, using it bit-by-bit in everyday work, and constantly pondering of bigger, higher level systems. Finally I've taken the last four years to explore the possibilities for such systems and would like to share some results with you.

One aspect which I feel might be interesting to the community (and one in which my approach seems different from many projects presented here) is the choice of tools: Houdini as the main platform and Fusion for supplementary 2/2.5D work. Both are high level and high end node-based DCC packages. This gives me tons of freedom and allows to convert existing knowledge and understanding of computer graphics into the code most seamlessly. I've covered the approach in more detail here <http://www.the-working-man.org/2017/04/procedural-content-creation-faq-project.html> — the same article also discusses my first seriously big result — *Project Aero*.



Project Aero is a procedural aircraft design toolkit — a CAD system I developed in early 2015 as a proof of concept; an exercise in applying procedural approach to a less typical task than a landscape or city generator. The system allows a new, highly-detailed 3D model to be created in a couple of hours from scratch; no data is sampled from disk — every piece of geometry and textures are synthesized to fit the particular design. All models come out pre-rigged and ready to render, with parametric aging, non-identical symmetry, and unique copies on a button.



Though it had started as a tech demo, *Aero* has proven to be production-ready and served as the core technology behind *Flight Immunity* — an aircraft concept art project currently counting over 50 original designs and otherwise hardly possible within the same time frame.



While this first project has aimed to prove that such procedural enterprise was at all possible, the second one was meant to show that everything is. I've been intentionally seeking to make it as different from *Aero* as possible — in subject, techniques and any other aspect. *Kozinarium* is a creature generator; a true generator this time: a new creature can be created in one click (though CAD functionality is still there and every design parameter can be overridden manually). Several seeds can completely define both the shapes and animation for a new animal; a rig is generated procedurally too. Models are created with volumetric representations, animation largely relies on Finite Elements Model dynamics, and most of the shapes and motion paths are literally drawn with mathematical functions. I've covered *Kozinarium* as well as some more exciting stuff here <http://www.the-working-man.org/2018/04/procedural-bestiary-and-next-generation.html>.



These, and other endeavors of the last years, have left me with two outtakes which I'd like to share. One is that visual art is formalizable and can be expressed algorithmically to a much higher degree than it's commonly considered. This is the message going throughout all

my procedural work. It takes a lot of knowledge, it takes a lot of effort and devotion, it does take a vision, but it is formalizable. I found that while designing procedural systems I think in altogether different terms than ‘pixels’, ‘polygons’, or ‘UVs’. Translation into this technical language of CGI happens quite late in the process, but is already being done to some extent for each project. What’s needed is to generalize it.



For example, the basic notions like point, curve and surface can all be redefined at a higher abstraction levels, so that particular CG implementations like NURBS or polygons, raster or vector should become secondary and be derived from those. This in turn can form a backbone for a much more content-aware design system, “the generator of generators”. As a third and final project, I have actually conceptualized a point-based solution for this new system — a universal high level framework for procedural content creation. It’s sitting there as a set of notes in the drawer waiting for the proper funding.

The second outtake I've got from the journey looks a bit less optimistic — I'm observing exciting procedural things happening simultaneously across several different fields of knowledge, yet these fields seem to show little interest towards each other, each living in their own bubble. Film vs games, artists vs programmers, academia vs practitioners — only a few camps that quickly come to mind, each being involved with proceduralism in its own way but with a clear lack of dialogue in between. Hope this text might become a part of the change.



More words and pictures are available at www.kozlove.net

