Seeds
Issue 2

Credits

Editors: Jupiter Hadley - @Jupiter_Hadley
Joseph Brown - @jb03hf

Contributors:

Davide Aversa
Heather Robertson
Davide Prati
NoobStudios
Zachary Spector
Quantum Potato
Serin Delaunay
Łukasz Hryniuk
Rick Hoppmann
Joseph Brown
Valtchan Valtchanov
Luke O'Connor
Tom Coxon
Ben Samuel
Pavel Oreshin
Marcos Donnantuoni
Matthew Keff
Elle Sullivan
Peter Christian Jørgensen
Mike Cook
Marco Scirea
Matthew Santacroce
Mark Rickerby
Brandon Yu
Gillian Smith
James Earl Cox III
Jasmine Otto
Joseph Alexander Brown
Adam Summerville
Ahmed Abuzuraiq
Paul Jeffries
James Ryan
Kate Compton
Ethan Edwards
Munir Makhmutov
Andrei Gusev
Jason Grinblat
Heather Kelley
Audrey Moon
Owen
Hamna Aslam
George Baron
Isaac Karth
Bulat Lutfullin

Organisers:
Azalea Raad, Joseph Brown, Jupiter Hadley, Michael Cook, Rachel Hwang

Art Packs By: Ajay Karat AKA Devil's Garage, Cryoclaire

Speakers: James Ryan, Jupiter Hadley, Kristin Siu, Mariano Merchante, Mitu Khandaker, Rachel Hwang, Tyriq Plummer

Tutorials By: Bruno Dias, Christer Kaitila, Joseph Parker, Melanie Dickinson

Thanks to: Alex Champandard, Ben Porter, Blanca Pérez-Ferrer, Gabriella Barros, Innes McKendrick, Jo Twist, Katie Rose Pipkin, Mitu Khandaker, Phoenix Perry, Rami Ismail, Simon Colton, Strangethink, Tom Betts

Cover Art by: Matthew Keff

Some header/footer patterns from La Boite à Tortue's Procedural Tileset Generator:
https://tilegenerator.tumblr.com/

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.

Abian Hernandez
Adam Hill
Adam M. Smith
Adam Norton
Adam Summerville
Alden Etra
Alessandro "NeatWolf" Salvati
Alex Swaim
Alexander Zook
Alistair Roche
Andre Haensel
Andrew Armstrong
Andrew Kuntz
Andrew Lim
Andrew Maxim
Andrew Plotkin
Andrew Sutherland
AngoraFish
Anita gray Saito
Anne Sullivan
Ashley Elsdon
Åsmund Aqissiaq
Arild Kløvstad
Atul Varma
Aviv Beeri
Belchingcultist
Ben
Ben Brooks
Ben Bruce
Ben Lambell
Ben Wells
Benjamin Vance
Billmaya
Bob Culley
BrettW
Brian Dysart
Cameron
candeira
Cara Warner
Carol Beck
Charles Tangora
Charlie Croft
Chip Lynch
Chris Janes
Chris Knight
Chris S
Chris Welch
Christiaan
Christoffer Holmgård Pedersen
Christopher Mangum
Christopher Weeks
Chrisx2ds
Ciro Durán
Claire Blackshaw
Corey Farwell
Coyan Cardenas
Csongorb
D.Rail
Daimadoshi_CL
Dan Duggan
Dana Chayes
dananddna
Daniel Currie Hall
Daniel Gonçalves
Daniel Rehn
Daniel Wildschut
Darkliquid
Darren Grey
Dave LeCompte
Dave R
David
David Berghoff
David Pittman
Davide Aversa
Delibean
Douglas Gregory
Drew Petersen
Dushanth Daniel Ray
Dylan
Eclogiter
Eijmans
Eleanor
Elliott Draper
Emil "AngryAnt" Johansen
Emil Ng
Emily Short
Emil Ng
Emily Short
Eric Schwarzkopf

Ethan Edwards
Evan Cobb
Evan Jones
Fausto Fonseca
Fed Kasatkin
Feufochmar
Francis Fitzgerald
Frank Lyder Bredland
GapGen
Gareth Cadman
Gary Steinke
Gautier Nadé
Gavin Inglis
Generesque
George Koutsikos
Giles Richard Greenway
Gillian Crowley
Gillian Smith
Guest 442277269
Guilherme Tœws
Guillaume
Hector Dearman
Henry Lisowski
Hodge
Ian Fagan
Ian Horswill
IdleDice
Isaac Fulkerson
Isaac Karth
Isquiesque
Jack Everitt
JakeTU
James
James Allenspach
James Coote
James Ryan
Jamie Woodcock
Jason Grinblat
Jeremy Apthorp
(nornagon)
Jim Whitehead
Jo
Joel Davis
Joerg Reisig
John
John Hattan
John Kane
John Oliver
John W. Bruce
John Watson
Johnicholas Hines
Jon South
Jonny
Jose Lema
Joseph A Brown
Joseph Carter Osborn
Joshua Sharp
JReynolds
Julien Delezenne
Jupiter Hadley
Jurie Horneman
Justin
Justin Loudermilk
Kate Compton
Kaydiv
Kayn
Kemp
Ken Gagne
Kevin O'Neil
Kirsty
Kitfox Games
Konstantin Kitmanov
Kris Loukas
Lee woo yeon
Liza Daly
Llaura
Long2Wed Marriage
Services
Lucile
Luke Miller
Luke Miller
Lynn Cherny
maetl
Malena Klaus
Marc Destefano
Marco Scirea
Mark Fletcher
Mark Gritter
Mark Renner
Martin O'Leary
Martin Randall
Matt Walsh
Matthew Ahrens
Matthew Crossley
Matthew Guzdial
mattperrin
Michael Coulthurst
Michael Gradin
Michael Nørskov
Mike
Molly O'Brien-Manley
Monica Neddi
mono
Morteza Behrooz
Nat

Natalie Freed
Nate Marsh
Nathan Pride
Nathaniel Mitchell
nemmons
Niall
Nicholas
Nisse Hellberg
Noah Benjamin Maze
Noah Swartz
Pablo Farias Navarro
Pat Ashe
Patrick Ewing
Patrick Reece
Paul
Paul
Paul Sottosanti
Peter B
Petri Nakari
philnelson
Pier Luca Lanzi
Pierce Brooks
Prad Nelluru
quantumpotato
Quinn Monk
r618
Rachel Hwang
Rebecca Paliwoda
RevDanCatt
Richard Alison
Richard Gillen
Richard Newbold
Rob Homewood
Robert Kohut
Robert Masella
Robert Turner
Robin Baumgarten
Robin Todd
Rodolfo Rosini
Rogelio E.
Cardona-Rivera
Ronny Anderssen
Rune Skovbo Johansen
RustyHarper
Sally Kong
Sam
Scott Grant
Sean LeBlanc
Sean Marzec
Sean O'Rourke
Several
Shaddock Heath
Shawn
Shawn Graham
Shobha Kazinka
SirPsychoBexy
skeptic
T. A.
taurisince1983
Tero Parviainen
Thomas Smith
Tieg Zaharia
Tim Stoddard
TJ Houston
Tom Howard
Tommy Thompson
tor gausen
Travis Fort
Trevor Adams
Tristam MacDonald
Troy Humphreys
Tyler
Valentina Lore'
William Hensley
Young-hoon Lee
Zachary Spector
Zack Johnson
Zebulon Pi
ZiJing

& everyone who shared our Kickstarter and who told a friend about it.

ProcJam

Make Something That Makes Something

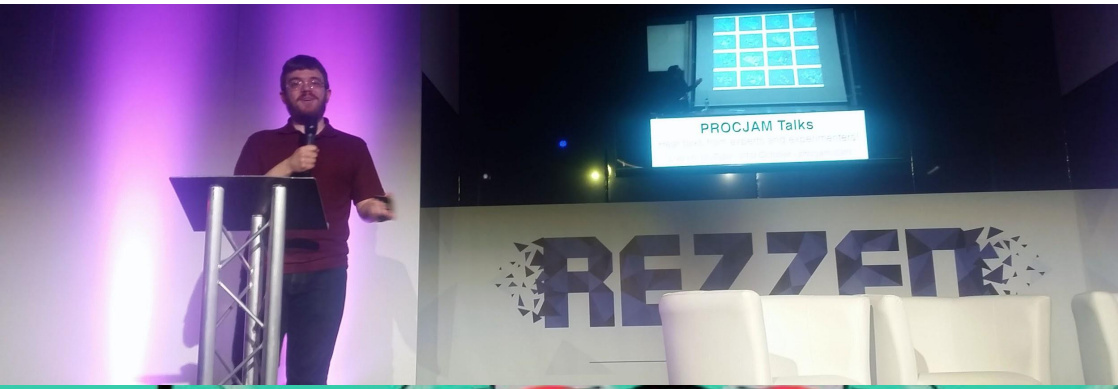
The ProcJam is a unique, relaxed game jam that aims to make procedural generation accessible and to show off projects that are pushing the boundaries of generative software. As a whole, this jam is laidback, easy to enter, and fun to be apart of. We are working to build a community of friends and peers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



PCG in my Business Card

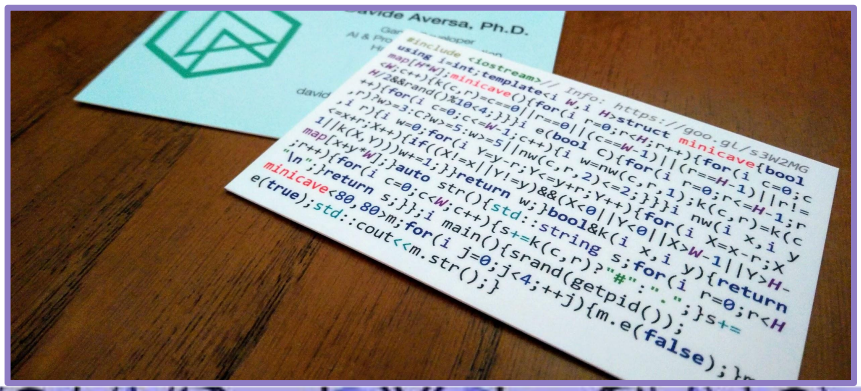
By Davide Aversa
@thek3nger

I like PCG, and I like having it with me all the time! For this reason, I wrote a cellular automaton based procedural caves generator algorithm that fits in a business card. I didn't write an algorithm to generate design for my business cards, that would be unoriginal (but cool). I wrote **the algorithm** on them.

"I didn't write an algorithm to generate design for my business cards, that would be unoriginal (but cool). I wrote the algorithm on them."

The algorithm itself is not new and I have already written it many times. Cellular Automata are my personal approach to "Hello World": when I want to try a new language, I write a PCG algorithm in it.

The idea started from an article by Matt Zucker, see https://mzucker.github.io/2016/08/03/miniray.html, in which he describes his challenge of writing a ray tracer algorithm that fits a business card. I thought it was cool. Even if I never got into code obfuscation challenges - it makes my OCD cringe and I lay down in pain - I thought it was a nice idea for a business card. I wanted to try. Of course, I could not do the same with a ray tracer algorithm; I already had my problems with the non-obfuscated version. Then I remembered that I already had a C++ code doing something I like (Procedural Content Generation, indeed) that is a good candidate for this: my cellular automaton code.



```
using i=int;template<i W,i H>struct minicave{bool
map[H*W];minicave(){for(i r=0;r<H;r++){for(i c=0;c
<W;c++){k(c,r)=c==0||r==0||(c==W-1)||(r==H-1)||r!=
H/2&&rand()%10<4;}}}i e(bool C){for(i r=0;r<=H-1;r
++){for(i c=0;c<=W-1;c++){i w=nw(c,r,1);k(c,r)=k(c
,r)?w>=3:C?w>=5:w>=5||nw(c,r,2)<=2;}}}i nw(i x,i y
,i r){i w=0;for(i Y=y-r;Y<=y+r;Y++){for(i X=x-r;X
<=x+r;X++){if((X!=x||Y!=y)&&(X<0||Y<0||X>W-1||Y>H-
1||k(X,Y)))w+=1;}}return w;}bool&k(i x,i y){return
map[x+y*W];}auto str(){std::string s;for(i r=0;r<H
;r++){for(i c=0;c<W;c++){s+=k(c,r)?"#":".";}s+=
"\n";}return s;}};i main(){srand(getpid());
minicave<80,80>m;for(i j=0;j<4;++j){m.e(false);}m.
e(true);std::cout<<m.str();}
```

It is not near as complex as a ray tracing algorithm but it was fun. The final code is shown above. It simple and unpolished but it is functional and modular too! You can see an example output in the images.

Bot Merit Badges

By Kate Compton
@GalaxyKate | http://www.galaxykate.com/

I MADE A BOT
AND MY BOT REMIXES MEDIA

I MADE A BOT
AND MY BOT SPEAKS TRUTH

I MADE A BOT
AND MY BOT GROWS LIFE

I MADE A BOT
AND MY BOT TELLS STORIES

I MADE A BOT
AND MY BOT CREATES WORLDS

I MADE A BOT
AND MY BOT IS DEEPLY CONFUSING

I MADE A BOT
AND MY BOT MAKES RECIPES

I MADE A BOT
AND MY BOT TELLS JOKES

Serious Generation for Social Good

By Hamna Aslam & Joseph Alexander Brown
@Hamna72 & @jb03hf

According to the United Nations Refugee Agency (UNHCR), on average one person is forced to flee their home every three seconds due to war [3]. Presently, relief camps are the only hope of shelter for majority of these people. Relief camps are not only a roof for temporary shelter, they have become fully organized management systems.

The design of relief camps is a constrained optimization problem with high dynamicity in constraint variations. We have developed a generator for relief camp designs which takes a list of relief camp utilities as an input, and finds the sub-optimal design within the specified constraints.

"The camps designed by the generator are compared to the designs generated by humans via a game."

The generator utilizes a Genetic Algorithm which places tents, water sources, and toilets on a two-dimensional space and checks them against the requirements provided by the World Health Organization (WHO) and the UNHCR for humanitarian response standards [1,2]. The generator provides the fastest way to see multiple designs against the same settings of resources and the most feasible one can be picked to be deployed.

Presently, the generator includes tents, toilets, and water containers to build camps. The camps designed by the generator are compared to the designs generated by humans via a game. The relief camp manager game allows a player to set up the camp according to their preferences and the fitness formula that has been implemented is also used as a feedback for the player as a score. Figure 1 shows a screenshot from the relief camp manager game. The human player can generate designs in the game and can compare the effectiveness of their designs against the software-generated camp designs for the same number of resources. The human solution can then be optimized by this comparison.

We have run human competitive tests where human players have designed relief camps via game and the generator has also produced designs for the same settings and number of resources. The results showed that the generator outperformed human players when the designs were compared according to the standards from the WHO and UNHCR.

Necessities	Provision Guideline
Drinking water (per person)	3 litres per day approx.
Tent to water point distance	500 metres (max)
Distance between toilets and tents	50 metres
Distance between toilets and water container	30 metres (min)

Table 1: Guidelines and figures by WHO and UNHCR implemented in the game [1,2]

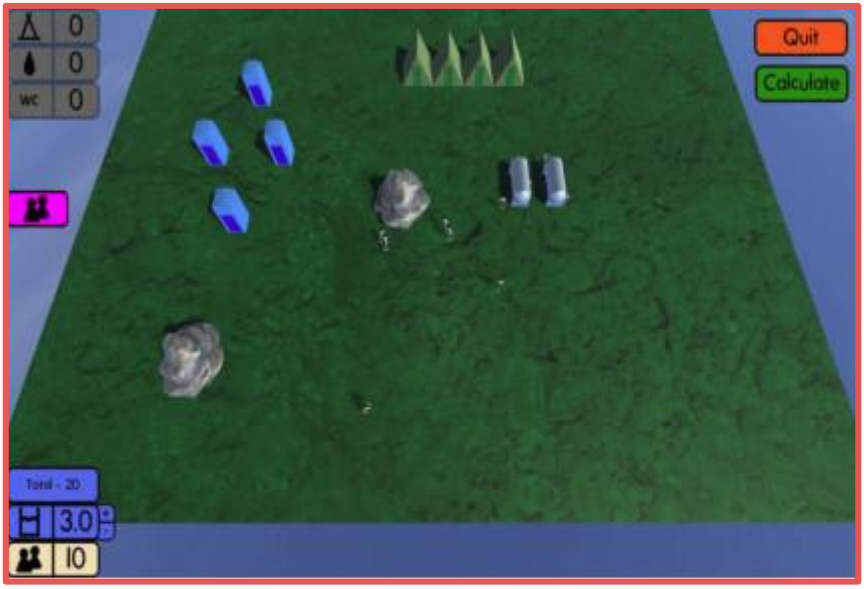


Figure 1: Camp setup Interface

As the world is facing increasing number of displacements and global refugee crises together with uncertain and limited resource availabilities, finding the best possible designs for camps is a challenging issue. The generator generates designs in a short period. Through the relief camp game, camp managers can compare camps according to their priorities. Both solutions generated by the human player and the generator can be combined or compared to get the best possible outcome.

"The motivation behind our work is to improve the relief camp design process"

The motivation behind our work is to improve the relief camp design process by developing a methodology that can help in generating camp designs which are as near to the humanitarian standards as possible in the presence of limited facilities and resources.

For future work we are looking forward to considering more utilities of relief camps in the generator as well as in the game. More information about our work can be found in [4].

References:

1. Reed, R., Godfrey, S., Kayaga, S., Reed, B., Rouse, J., Fisher, J., Vilholth, K., Odhiambo, F.: Technical notes on drinking-water, sanitation and hygiene in emergencies (2013).

2. UNHCR: Comparison of humanitarian standards, the sphere project and UNHCR emergency handbook (2001). Accessed 26 Oct 2016

3. Yonetani, M., Lavell, C., Bower, E., Meneghetti, L., O'Connor, K.: Global estimates 2015, people displaced by disasters. In: IDMC, Internal Displacement Monitoring Centre (2016). Accessed 6 Oct 2016.

4. Aslam, H., Sidorov, A., Bogomazov, N., Berezyuk, F., Brown, J.A.: Relief Camp Manager: A Serious Game using the World Health Organization's Relief Camp Guidelines, Applications of Evolutionary Computation (2017), pp. 407-417. http://tinyurl.com/reliefcamp

Life Simulator Engine

By Zachary Spector
@LogicalDash | https://cybre.space/@LogicalDash

Life simulation is a natural genre for games that use procedural generation. There's really no limit to how much of the world the player might see, or how many systems they might interact with; if the player can't understand it all, and has to fumble through with gut feelings and best guesses, that's verisimilitude! But it's not such an accessible genre for developers, for the same reasons. Life sim games are as complex as roguelikes, but with much more persistent world state to keep track of, and it's not quite so easy to get the player to accept any breaks with reality, because it's such a familiar reality.

I was playing Dwarf Fortress, the most visibly "proc gen-ish" example of the genre, and getting frustrated at some of its technical shortcomings. First the interface, which others have complained about at length, but then, there were lots of dedicated modders trying to improve that.

Why should Toady worry about the interface when the part he's actually good at is the world model? I'd have been perfectly content to just play a modpack if I didn't have to wait months after every major release. So...why *did* I have to wait? Why, when it appeared that most of the established systems were unchanged, did the modders have to redo everything? Building my own modded copy of the game provided an answer: the game is only moddable insofar as it stores a lot of things in configuration files, and lets people change those. For the really useful stuff, like the graphical frontend Stonesense, a dedicated team of modders had to hack the game and make their own programming interface (called DFHack) for other modders to use. That's a huge amount of work. It's no wonder Stonesense only recently got to the point where you could really play with it.

"...it's not quite so easy to get the player to accept any breaks with reality, because it's such a familiar reality."

I think it's a shame that Dwarf Fortress is so inaccessible to both players and modders. I want there to be more games like it, but I understand why there aren't; its development consumes Toady One's whole life, even with all the help from modders and players. Toady doesn't want to release the game's source code, even though

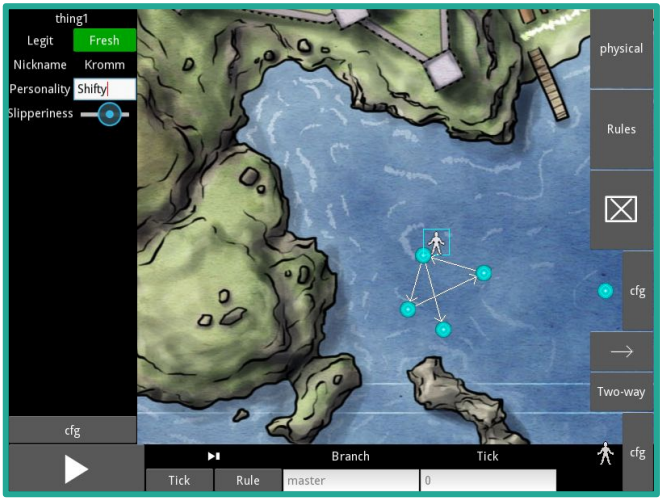


Image of ELiDE, the graphical frontend

the modders know all about its inner workings; and doesn't want to make an official programming interface, because that's a lot of work. So the game's mod scene proceeds in this haphazard way, with so much wasted effort, to make the game marginally playable for newcomers. It's not very surprising that there aren't many games like this!

Maybe if it were easier to mod similar games, there would be more of them. So I'll make that happen. The Life Simulator Engine, abbreviated "LiSE" and pronounced "lies," is not a game engine in the traditional sense. It does not have any graphical interface, although I am building such an interface for it;
LiSE itself is a rules engine, which keeps snippets of code describing things that can happen and conditions they should happen in, and applies them to a world model based on directed graphs.

You can simulate a world in this engine, and modders can use the same programming interface you did, even if you don't want to release your code; LiSE can run as a web server, and modders can build frontends for its RESTful web interface. But if you do release your code, the composable nature of the rules means that they will work in some capacity without alteration in any other game made in LiSE.

I anticipate that casual developers will make cool stuff by playing "mod tennis," adding a rule and emailing the game to a friend who does the same and sends it back. Like Photoshop tennis for proc gen. Maybe, someday, there'll be enough games in the engine that you can make a new one by reusing rules from old ones, and not write any new code at all. The interface for the world model is based on that of the NetworkX graph library, which has lots of functions to generate and mutate graphs in ways that I don't understand. I've strung them together into a little syntax based on method chaining that seems like it could be fun to play around with, but I've only used it to make some grid-shaped graphs and some boringly chaotic ones. The project would benefit greatly from people doing random wacky stuff with it and giving feedback. The documentation is lacking at the moment, and I'm putting off writing it until I finish an overhaul that allows the engine to pause in the middle of a turn, a feature that I only realized was important when I tried to build a system to make the frontend present menus to the player.

LiSE does not use save files in the traditional sense. Every change to the world gets journaled to a database, currently SQLite, though support for other databases is planned. This allows developers to see everything a player ever did, and every change that was ever made to the world, and to rewind and replay all that in the graphical development kit. This will aid in debugging, and may provide for some new gameplay mechanics too -- the player can travel through time easily, if you want to let them. Eventually you'll be able to query the past, to answer questions like "When were you last here?" and "How many times have you eaten spaghetti?" without having to write any special code to track those things aforethought.

"The project would benefit greatly from people doing random wacky stuff with it and giving feedback."

Check out the LiSE GitHub repository: https://github.com/LogicalDash/LiSE If you're interested in LiSE, please consider filling out this survey: https://goo.gl/7N1TBj

Finding Room to Reflect in Generative Design: The Slow (proc)Jam

By Gillian Smith
www.sokath.com | @gillianmsmith

At the first procjam in 2014, I gave a talk titled "make something that makes something that isn't a game". I talked about games, but I also talked about why we could think more broadly about generative design outside of games. About what the future of PCG could be, and why that future is exciting, and what we need to be looking at to get there. I tried to argue for thinking about interfaces to PCG, and making tangible things, and making better design tools, and caring about people. Honestly, I was getting pretty burned out on games when I gave that talk.

"It pulled quotes and color palettes from the internet, generated random motifs, and made weird little art pieces."

I wasn't planning to make anything for procjam, but when I flew back home from Europe, I figured I'd follow my own advice, and used up my laptop's battery life trying to make something that makes something that isn't a game. And then I got home and stayed up way too late at night, feeling this urge to create. I hadn't felt that feeling in years.

The project was a cross-stitch sampler generator called Hoopla. It pulled quotes and color palettes from the internet, generated random motifs, and made weird little art pieces. The best of its output looked like 19th century glitch art and the worst looked like a child's attempt at using MS Paint for the first time.

When procjam ended, I threw what I had up on itch.io, and thought I was done. But suddenly, during my downtime and especially when I traveled, I would pull out Hoopla and add new features until my battery went dead. And then, I began integrating what I was learning about generative textiles into my research – designing games that use crafting as an interface, treating gameplay as a generative system, and attempting to diversify games through craft.

Working on Hoopla here and there became a reflective practice for me – an escape from what I "needed" to work on that gave me a safe,

isolated space to explore new ideas. I wouldn't be where I am today without that little procjam project, sitting on the side, waiting for me to come and poke at it a little bit more, whenever I was ready. For a long time I didn't even think about completing it. I didn't think of working on it as making "progress". It was more about the slow process of creation, experimentation, and reflection.

This year, I decided I'd learned all I could from the digital experimentation. Every night in January, after my son had gone to bed, I parked myself on the couch and took out an embroidery hoop. My thoughts drifted as I engaged in the repetitive activity of stitching generated motifs. Stitch by stitch, pixel by pixel, I made my favorite designs from Hoopla live in the real world. I came to think of it as following a slow, reflective, and interpretive rendering process. Every decision I made in my code was slowly reflected, by hand, on a small piece of cloth that I stitched one square millimeter at a time. Every error I made in my stitching had to be either painstakingly undone, or left there forever.

When I finished my stitching, I literally shipped Hoopla: wrapped in bubblewrap, in an old Amazon box, it traveled across the ocean from my home in Massachusetts to an art gallery in Dublin. It was shipped back to me at the end of the summer, at the close of the exhibition: my permanent reminder of a beloved piece of software that had in the meantime already fallen to bitrot.

Erosion, Weathering, and History

By Isaac Karth
@isaackarth.com | @proc_gen | procedural-generation.tumblr.com

I was looking at some badlands recently. Not the famous South Dakota geologic formation, but a similar, smaller scale version of the same phenomenon, where the erosion has stripped away the soil and left very little for vegetation to cling to. Badlands leave bare the effects of erosion, making it easy to trace the paths that water and wind have carved out of the rock. The history of process is made visible.

"You can fake it: you can think of Caves of Qud's invented history as being like a model-maker's weathering."

Unlike pristine computer renders, real-world objects are imperfect, and part of those imperfections are the marks left by object's history. Model-makers often go out of their way to add weathering. The history of the object is recorded in its scuffs and stains.

Some generative techniques don't leave a history. Perlin noise, for example. But more simulationistic processes sometimes do create an actual history for an object. The early artificial-life experiment Sugarscape used agents to simulate a history. *Dwarf Fortress* simulates an entire world's history to give context to its generation.

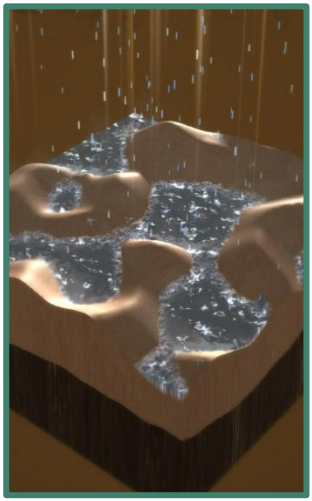
You don't need to actually simulate the history, of course. You can fake it: you can think of *Caves of Qud's* invented history as being like a model-maker's weathering. Most weathering techniques use fast physical processes to fake a lengthy history. Texturing techniques use mathematical cheats to do the same.

History is useful for more than just textures and terrain: many generated things can benefit from having the additional context from a past history. Character backstories are another obvious one. How a building has been used in the past affects its shape in the present: one perspective on this can be found in *How Buildings Learn* by Stewart Brand.

Once you have a history, consider using it as part of the presentation. Tell the user about it. This could be literal, such as the way *Dwarf Fortress* attaches little histories to the artwork on objects. Or it could be more abstract: perhaps pottery made in different towns use different subsets of the descriptive tags, and we can use the information about where and when it was made to add flavor and imply a past.

You can use it directly, by telling the player the lore, or find ways to do it indirectly. Environmental storytelling depends on giving the player a sense of past events solely through the present configuration of physical objects. If we have a history, we can use it to more effectively procedurally generate environmental storytelling.

Or, you can feed the history into other generative systems. Consider a system that generates legendary medieval weapons, with associated histories. The history of the object can be used to influence the visual appearance of the model: this owner carved runes in the blade, that owner wrapped the hilt in red leather, this chip is from when it was used to slay the dragon.



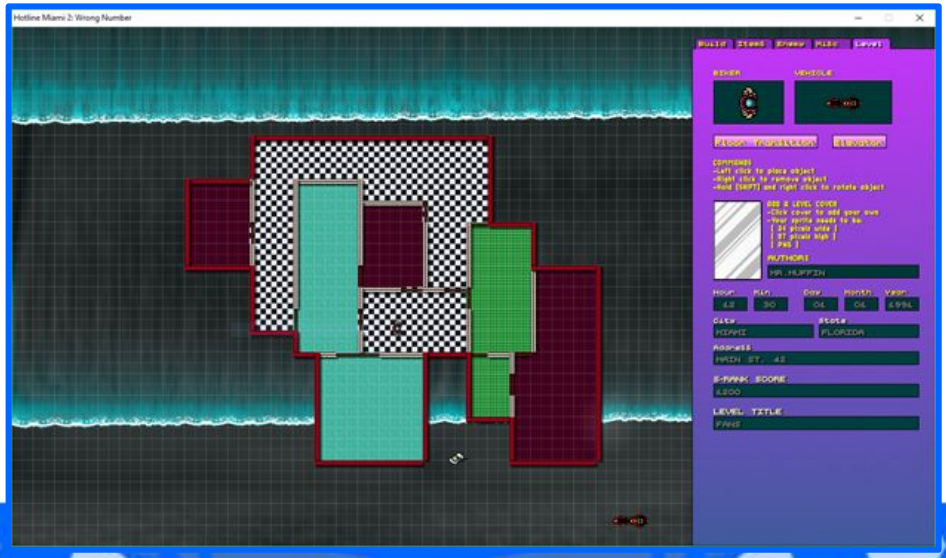
There are a lot of applications to invoking history in our generative processes, tying our generated artifacts into a past and a context. Literal and figurative weathering is useful for all kinds of generation, whether you fake it or simulate it.

Reverse Engineering of a Level Editor File to Allow for PCG Elements in Hotline Miami 2: Wrong Number

By Joseph Alexander Brown, Bulat Lutfullin, and Pavel Oreshin
@jb03hf

Hotline Miami is a series of 80s themed arcade shoot-em-up games. In early 2016, a beta level editor was provided for the second release in the series Hotline Miami 2: Wrong Number. In an effort to apply PCG elements, in this case levels, to a working commercial game, see [1], we needed to examine how the game stores levels. Unfortunately, while the level editor exists and humans can create new and interesting levels for it, PCG elements work in the level of the file, which are not human readable and which has no explicit mapping of the file to in game objects. In order to determine how to edit these files as process of reverse engineering was used in which known levels were first created by a human and correlated to the results seen in the files. This slow process allows for a mapping of elements of the file to elements shown on screen.

"This slow process allows for a mapping of elements of the file to elements shown on screen."



To understand file structure, we created empty level and made small changes to see the difference in files. The Hotline Miami 2 level is described by a set of six plain text files:

Level.hlm

Contains meta information of file such as level name, author name, size of the level, music id that will be played during game, etc.

Level.ver

Contains only one number - version of the Hotline Miami level editor.

Level.obj

Holds information about player character, player's car, doors, enemies and decorations. Each entity is described by its ID, coordinates (x, y), sprite ID, and rotation in degrees. Also, different types of objects have unique attributes like behaviour type for enemies AI (static, patrol, idle).

Level.tls

Holds informations about floor tiles like coordinates and sprite ID. Each tile is a square of size 16x16 px.

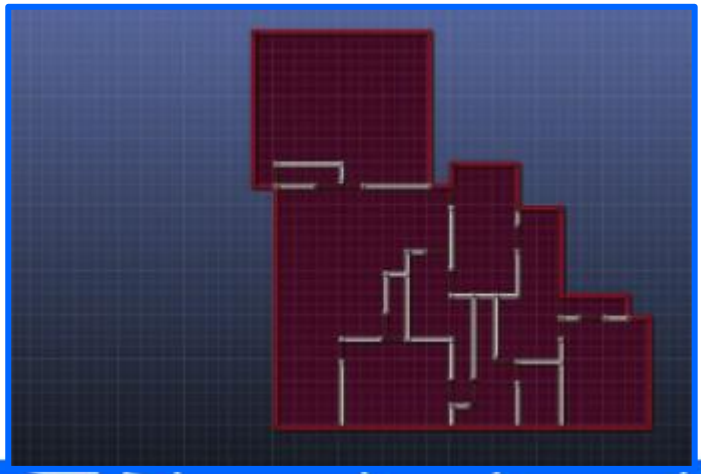
Level.wll

Contains description of wall segments. Each wall is assumed to be 2 tiles or 32px wide.

Level.play

This file is used in game mode as the final storage and contains all the information described in previous files. Level editor works without this file and is compiled when a save is made in the level in the editor.

Level editors such as those provided for Hotline Miami have been seen to have value for the developers, they provide extensions to the life of games and communities for players to trade, build, and play. We would encourage developers to not only provide editors which are usable by humans, but that allow for PCG developed levels to be implemented without the need for deciphering the files used for the game.



This could be accomplished by either using a format for the files which is human readable or giving a clear definition of how a level is stored, and by a listing of the available elements used in the game mapped to their unique id.

Future work is to continue our examination of the file structure to allow for the placement of objects and enemies.

References

[1] Joseph Alexander Brown, Bulat Lutfullin, and Pavel Oreshin. “Procedural Content Generation of Level Layouts for Hotline Miami”. 9th Computer Science & Electronic Engineering Conference, to appear, 2017. Preprint at: http://tinyurl.com/Seeds2-Hotline.

Physically Accurate Calendars and Astronomical Events

By Davide Aversa
@thek3nger

It is probably due to my details obsessed behavior, but every time I need to create a new world, I get lost into pages of physical and orbital calculations to find its most fitting calendar. In fact, I find it unbearable when distant planets or exotic fantasy worlds have a bland 12-months calendar. In the name of the Ancient Gods, how can a world with 5 suns and 17 moons have a year made of 12 months of 30 days? How do you define a "year" when you have 5 suns? What is even a "day" in that situation! On which satellite is based the lunar month that "casually" divide so perfectly your year in 12 intervals?

"It is a shame that we always use the same earth-like paradigm. We are limiting ourselves with just one of the possible outcomes."

These and many other are the questions I ask myself every time I forge a new world for my adventures. They seem useless picky questions or worthless details, but I strongly disagree. Calendars, seasons and astronomical events can shape entire cultures and are the most primordial shaping forces to which a species is subjected in its early years. It is a shame that we always use the same earth-like paradigm. We are limiting ourselves with just one of the possible outcomes.

Fortunately, when I write some short novel in my spare time or I design a D&D adventure, this is not a big deal. After all, once I world-built my setting once, I can spend years writing stories in there. So, this is a task I just need to do once in a while.

But when I move to game development and procedural content generation, I face the same problem multiple times per day. Now the task of designing a physically accurate calendar must be done every time the player clicks on "generate new world". I don't want to limit myself to the earth-like calendar again and,

surprisingly, there are very few games with such deep calendar generation. That's why I wrote a software helping me with this task.

In general, civilizations calendars are complex creatures. Without entering in many details, their generation can be seen as composed by two main blocks: the Celestial Mechanic Block and the Cultural Block.

I will not go over the Cultural Block for now. This is the step where we define names for seasons, months, days of the week and even the concept of "week" itself. It is an important step but not the step I am interested in. In fact, it depends on so many cultural and historical aspects that is almost impossible to design a general purpose procedural algorithm for it. Especially one that is independent from the actual setting of the game.

The interesting part is in the Celestial Mechanic Block. This part is concerned with the computation of the astronomical variables that are relevant to a calendar. That is, given some basic input data about the planet –such as planet distance from the sun, the number of satellites, the mass of the sun, and so on– we compute values such how long the planet takes to complete one orbit, the duration of each season (if there are any), the duration of a day, lunar phases, how to handle leap years, and more.

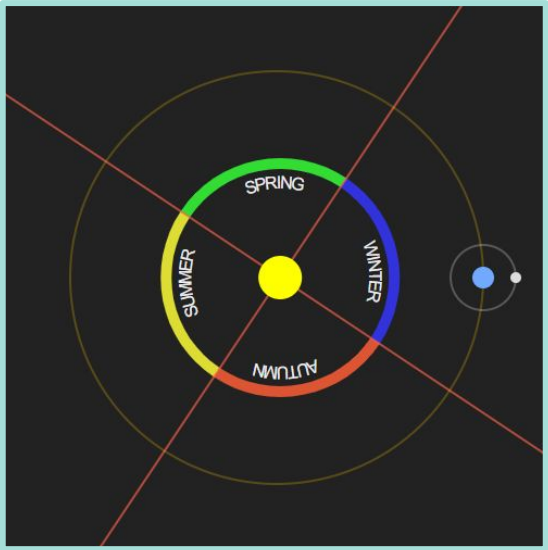
These values are the building blocks for each calendar and, most important, they are independent from the population culture/language. They only depend on the eternal dance of celestial bodies around the space.

Even if this seems simple as applying a bunch of astronomic formulas, there are still a lot of unanswered questions. When we move away from the standard sun-earth-moon scenario, things start to get very messy. In a binary star system, the duration of a solar day may

"In general, civilizations calendars are complex creatures."

become irregular, seasons may assume complete different meanings, they may be more than 4, or none at all.

Even when we have our standard sun-earth-moon scenario, things may be harder and interesting to explore. For instance, if the planet revolution is very long (e.g., an 80-months-long year) it is unlikely that the “year” will be the center of the calendar. Season may be a better fit in this case. Imagine the stories for a planet in which winters and summers last long for years (our earth years).



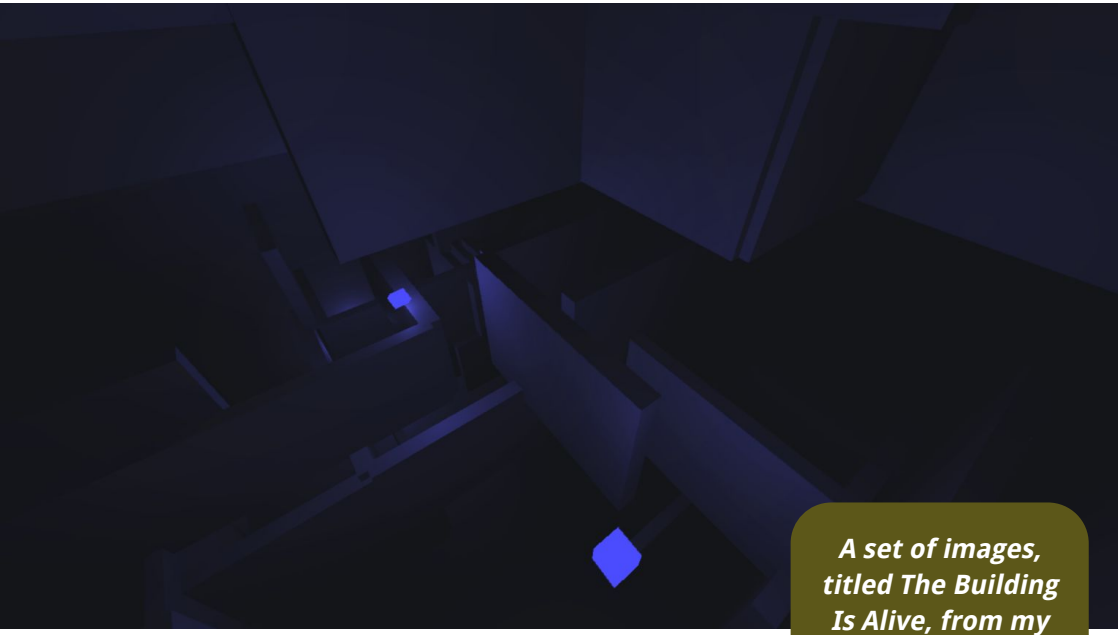
I love this kind of questions and that’s why I am investing some of my time into this procedural calendar generation. An early version of my attempt to grow calendars and astronomical events from orbital parameter can be found here.

I love this kind of questions and that’s why I am investing some of my time into this procedural calendar generation. An early version of my attempt to grow calendars and astronomical events from orbital parameter can be found here. For now, it works for earth-like planets and there is no fancy stuff.

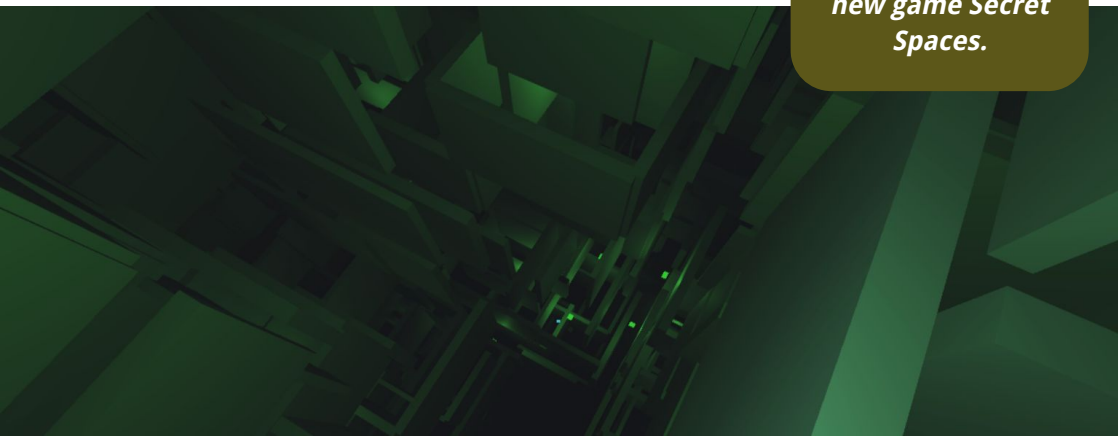
But I encourage you to visit my repository (https://github.com/THeK3nger/calendar-generator) and discuss many of these fascinating questions!

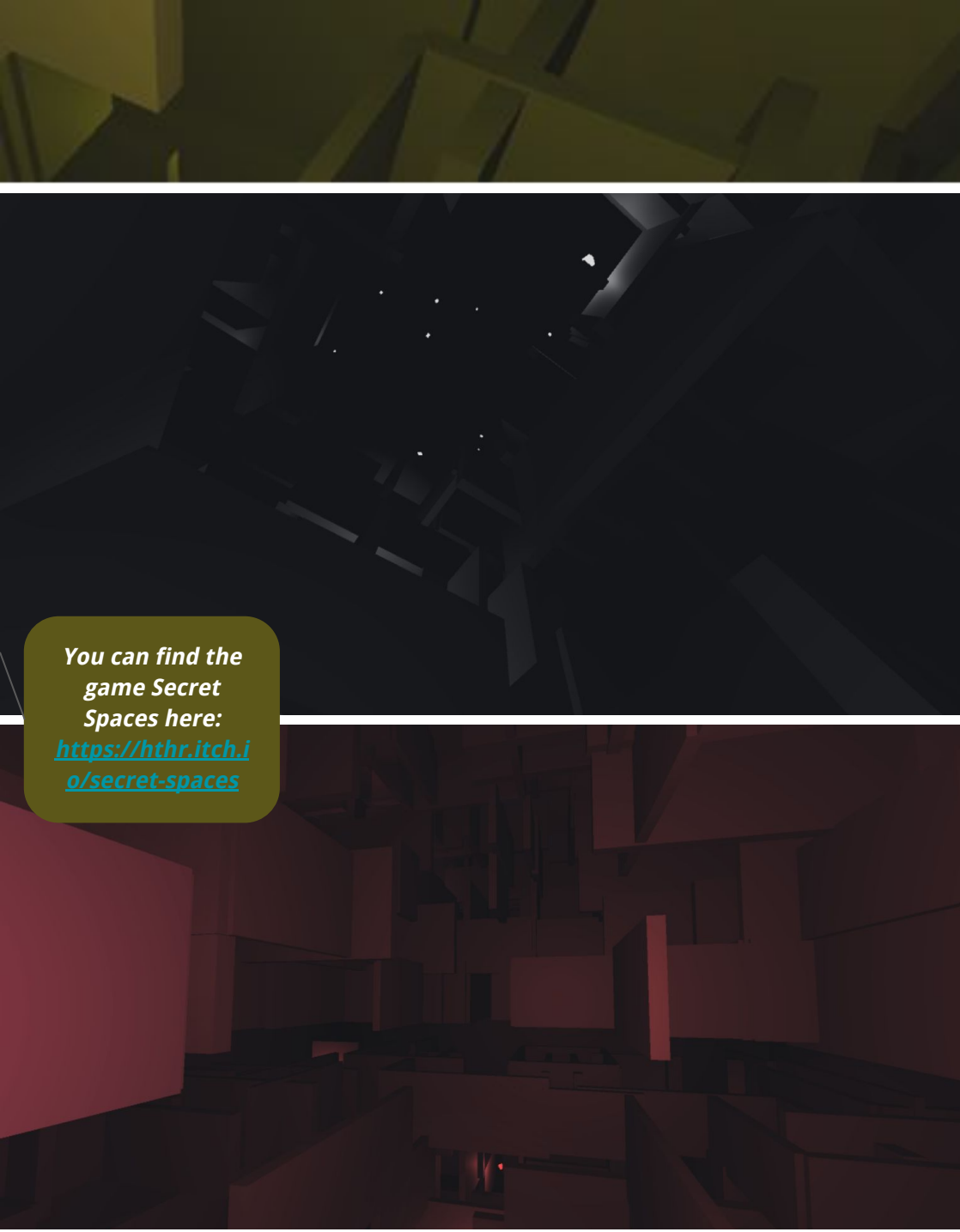
The Building Is Alive

By Heather Robertson
@HTHRFLWRS | heather.flowers



A set of images, titled The Building Is Alive, from my new game Secret Spaces.





You can find the game Secret Spaces here: https://hthr.itch.io/secret-spaces

Turing Patterns: On the Procedural Uses of Morphogenesis

By Elle Sullivan
@THISISDINOSAUR

Most people know Turing for his work in AI, codebreaking, or for Turing machines, but, as if he didn't contribute to academic imposter syndrome enough, he also produced seminal work in theoretical biology, with a single paper, entitled "On the chemical basis of morphogenesis". Morphogenesis is the biological process of how organisms grow into their specific shape.
Turing's contribution to this was a reaction-diffusion model of morphogenesis, describing the diffusion of two different chemicals, one activating growth and one deactivating it, decades before such a thing was actually observed.

$$u_n = \Delta t(D_u \nabla^2 u_{n-1} - u_{n-1} v_{n-1}^2 + F(1 - u_{n-1}))$$

$$v_n = \Delta t(D_v \nabla^2 v_{n-1} + u_{n-1} v_{n-1}^2 + (F + k) v_{n-1})$$

The discretized version of the Gray-Scott model for the simulator

When you hear the idea of a mathematical model of morphogenesis you may be lead to the idea of simulating such a system to grow a creature or a plant, but unfortunately this would require a significant leap in our current understanding. Morphogenesis' immediate interest to procedural generation is instead in some of the patterns it can produce, including different types of stripes, spots, spirals, and hexagons. These 'Turing patterns' can bare a striking resemblance to patterns found in nature and can look quite organic; one of the parallels of most obvious interest to procedural generation being the similarities to animal skins, such as leopard skin, the spots on giraffes, and zebra fish.

On top, the Gray-Scott model. On the bottom, the final equation that tells us when Turing patterns occur.

$$\frac{\partial u}{\partial t} = -uv^2 + F(1-u)$$

$$\frac{\partial v}{\partial t} = uv^2 - (F+K)v$$

$$\vdots$$

$$q_c^4 = \left(\frac{1}{2}\left(\frac{v^2 - F}{D_u} + \frac{F+K}{D_v}\right)\right)^2 > \frac{(F+K)(V^2 - F)}{D_u D_v}$$

It is possible to construct a simulation of such a theoretical chemical reaction (a 'reaction-diffusion' system, where local chemical reactions are transformed into each other, and diffusion causes them to spread out). This can be done on a grid (ideal if we're going to be producing pixel images from this), where the concentration of each chemical for a particular square is given by a partial differential equation, with a diffusion term simulating chemicals moving into and out of the square into neighbouring ones, and a reaction term representing the chemicals reacting to each other. A Turing pattern occurs when a homogeneous solution that is stable without diffusion becomes unstable in the presence of diffusion (i.e. if when the concentration of each chemical is the same across the entire grid the concentrations don't change if there is no diffusion, but if you add back diffusion, and make a very slight change to the chemical concentration somewhere, a Turing pattern will appear).

Contact me on twitter if you would like further details of the mathematics. If you find the maths intimidating, don't worry, it could be a lot worse.

Some unpleasant maths later will tell us what combinations of parameters and starting chemical concentrations will give us a Turing pattern if we colour each square based on the concentration of one of the chemicals (in this example, orange where u = 0, moving to white when u = 1). Plugging these into the simulator now gives us a wide variety of interesting patterns.

$$q_c^4 = \left(\frac{1}{2}\left(\frac{-\frac{1}{\epsilon}\left(2u + \frac{FV}{q+u} + \frac{FV(q-u)}{(q+u)^2} - 1\right)}{D_u} - \frac{1}{D_v}\right)\right)^2 >$$

$$\frac{\frac{1}{\epsilon(q+u)^2}\left(Fu^2 - Fq^2 - 2qu + 4qu^2 + 2q^2u - q^2 - u^2 + 2u^3 + 2Fqv\right)}{D_u D_v}$$

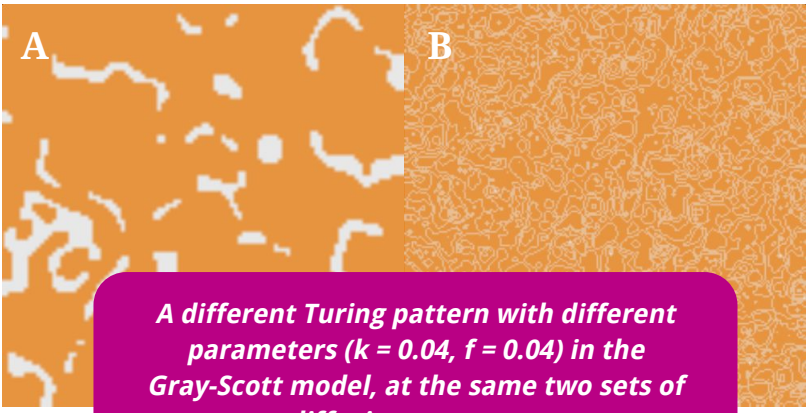
The equivalent for the Oregonator. See, I told you it could be worse.

One nice thing about this method of generation is that it works with a large number of different reaction-diffusion models. I use the Gray-Scott model here because it is relatively simple, the maths looks significantly less intimidating than some other models, and, despite this relative simplicity, displays a wide variety of interesting behaviour. I have also performed it with the Oregonator model, which models the Belousov–Zhabotinsky (or BZ) reaction, a nonlinear chemical oscillator, which can also have a wide range of exciting behaviour, such as propagating waves and spirals (and gets bonus points for being a real world reaction you can see). The mathematical procedure to find a Turing pattern and simulate it on a grid is the same for this or any reaction diffusion model, but the final equation that specifies the required parameters in this case is much more unpleasant looking.

A Turing pattern in the Gray-Scott model.

Both of these images were produced using the same parameter values (k = 0.047, f = 0.04) and show how you can change the scale by changing the diffusion constants.

A different Turing pattern in the Gray-Scott model.



A different Turing pattern with different parameters (k = 0.04, f = 0.04) in the Gray-Scott model, at the same two sets of diffusion constants.

Whilst this can produce a wide variety of surprisingly naturalistic patterns, one downside of this is the difficulty in predicting what set of parameters will give what pattern, making directed generation much more difficult, but at least the overall scale, and the colouration is controllable. All manner of different reaction-diffusion models could be used, and it can even be performed 3- dimensionally (so you could even perform the simulation across a 3D elephant shaped grid and could then extract a texture, removing the issue of lining up the texture with a uv map), although the real strength is being able to generate a wide variety of naturalistic patterns with very little effort.

A comparison of zebrafish and a Turing pattern. Taken from Advanced Reaction-Diffusion Models for Texture Synthesis , Sanderson et al., 2006.

Do not fear procedural puzzles

By Marcos Donnantuoni
@marcos_don | https://marcosd.itch.io

I love working with computers; not so much because of their abilities, as speed or memory (which are certainly useful), but because of two main disabilities: they cannot be frustrated or bored. That makes them ideal explorers of quasi-infinite spaces, which are usually forbidden to us. They can attempt for hours or days to construct objects that satisfy completely arbitrary criteria, and will not complain when that treasure they just discovered buried under billions of candidates is rejected by an aesthetic whim, or we change a line of code and send them again in a deep search perhaps destined to fail.

"That makes them ideal explorers of quasi-infinite spaces, which are usually forbidden to us."

But this lack of boredom or frustration is also dangerous, especially during the procedural generation of levels for puzzle games, which should not bore players with too easy levels nor frustrate them with too difficult ones (at least not too soon). How can computers, so oblivious to these dangers, avoid them?

Short answer: they cannot. But there are longer answers. An obvious one is to sort the puzzles by difficulty before presenting them to the player; this way, a player's progress in their "easy" zone will be quick and perceived almost as a tutorial, and the most difficult and transcendent puzzles will appear only when the player has "earned the right" to play them, after beating the intermediate ones.

Of course, this method has two problematic aspects. On the one hand, the very ordering criterion is necessarily arbitrary, and could work only for a subset of players (those more psychologically similar to the game designer/programmer); on the other hand, the classification of puzzles usually takes much longer than their generation, and may be too demanding for smooth gameplay in low-end computers or mobile devices.

The first problem is almost inevitable: different kinds of games are

enjoyed by different groups of people; we can embrace that without fear.

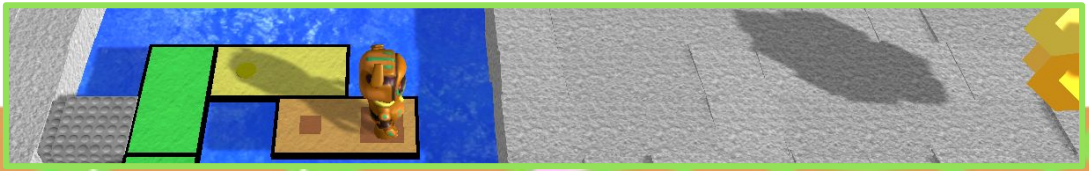
The second problem can be solved with a little cheating, by pre-generating and sorting the puzzles, and packaging them in the game (fortunately, they take up much less space than other assets, like textures or 3D models).

I use this method in the game I'm currently developing (codename "Pontibus"; I may present a work-in-progress version during ProcJam 2017), and it has yielded quite acceptable results.

In Pontibus the puzzles are combinatorial: players operate on certain objects until they arrive at certain winning configuration. This allows for two difficulty metrics: the puzzle's visual size and shape, which affect player expectation of its difficulty before solving it, and the puzzle's state-space vastness, which presumably reflects its real difficulty (this assumption is very debatable, but simple enough).

The first metric is easy to implement: it's basically a linear function of the quantity and size of moving elements of the puzzle.
The second metric is more laborious; it involves generating the entire "move tree" (or a sizeable part of it) and measuring its depth and frondiness (branching factor). If the proportion of winning configurations is high, the puzzle will score less at difficulty; similarly if the length of solutions is short.

This core technique can certainly be decorated with all manner of additional criteria, specific to each puzzle rule or set of rules; but it's surprisingly powerful as it is.



Who Needs Level Design? Using Player Input as Your Procedural Seed

By Quantum Potato
@quantumpotato

Hi Procjammers, I built www.QuantumPilot.me, where player input seeds the enemy AI's moving & shooting. No, the AI doesn't "learn" from the player nor does it compute a good defense against their playstyle a la Warning Forever [1], the AI simply copies your inputs and mirrors them back at you. The emergent result is fantastically unique levels for each individual player & play session.

"The emergent result is fantastically unique levels for each individual player & play session."

First I'll talk about the seed for this game and why it resonated with me. Next, the emergent gameplay benefits. Thirdly, challenges & quirks. Lastly, a brainstorm & challenge for you to use this in your own games.

SEED

I played ABA's [2] DefeatMe! about 8 years ago. Simple move and shoot with clones of you. I loved the concept but the implementation bugged me. The levels always repeated when you died - making getting stuck likely. Weapon spread and ship movement speed were procedurally generated (by modding the level #). This made for unpredictable gameplay as did the enemy ships warping back to the start when they finished their path. Lastly, the Deadline (A wall of bullets acting as a timer for the level) moved way too fast for comfort - little time to prepare or react. I fixed these design flaws in Quantum Pilot for a smoother experience.

EMERGENT GAMEPLAY

Most shoot-em-ups and "bullet hell" games feature geometric bullet patterns and the precise timing to weave between them as their core gameplay, with shooting mostly an afterthought. Quantum Pilot features The core dynamic gameplay & difficulty between the 2 skills of shooting & dodging.

If you shoot many times, your current level gets easier because there are more hitboxes for enemies to hit. Subsequent levels get immensely tougher by demanding better dodging. To make dodging easier you must shoot very accurately. This makes the clones shoot less and it's easier to skirt their bullets.

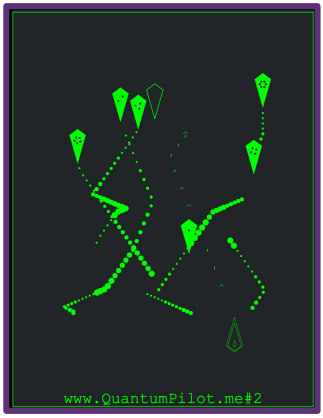
Shooting accurately is very tough and there's a risk mechanic: The longer you go without shooting looking for a clean shot & then you miss - you must act quickly to survive! This usually means reverting from "Shoot Accurately" to "Spray and Pray" - switching the skill requirement next rounds from easy dodging to hard dodging.

QUIRKS

Originally I had the player ship & enemy ship in Quantum Pilot start lined up evenly just like in Defeat Me. This led to what I call "the memorization problem" - the answer to a level becomes a puzzle where the player can learn & memorize the correct inputs instead of relying on good tactics. This isn't what I intended so I started randomizing the Clone starting positions. I also kept the player's X position the same between each level and when restarting the game -- there is no transition and from this seamlessness players often keep playing from the immersion.

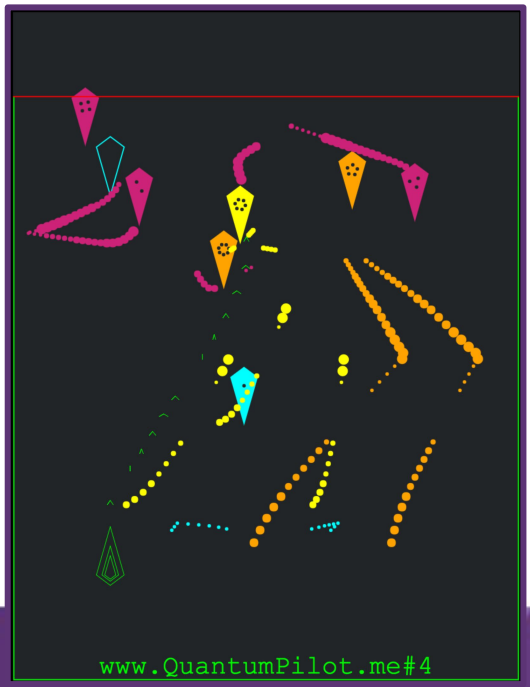
"This isn't what I intended so I started randomizing the Clone starting positions."

The Y position of the Clones & the Player is reset each round, and that's only to protect the player from being too close to the enemy on the next level and being instantly killed, or from shooting earlier than the Clones are and killing them too easily. Shooting earlier & earlier each successive round is a great strategy - a little too good and deemphasized tactical movement. I confronted this by having the Deadline speed up if you aren't shooting. This puts the pressure on the player!



I experimented with horizontal screen wrap (with

bullet + player ship projections so you can see where you're going). Most players liked this - it "looks cool" and it feels good to run away. But this literal lack of center destroyed the dodging gameplay - you could always just flee to one side, never worrying about being trapped. I removed this and put in a solid line for the border which visually frames the game much better and make choosing to weave into a bullet pattern a more interesting tactical decision.



FUTURE

Quantum Pilot is a simple exercise in replaying player input. You could do more - world generation based on what the player moves towards. Actual computation for bullet patterns that hit the player's movement patterns. Physics puzzles generated to deny the same quirks exploited previous rounds. A text-adventure where your clones help you. A 1v1 squad FPS where each team gets an AI with the movement copied from their opponent. What can you do?

SECRETS

I encourage you to make learning to play masterfully the most satisfying secret in your game. When you are smiling from gameplay, commit your code.

REFERENCES

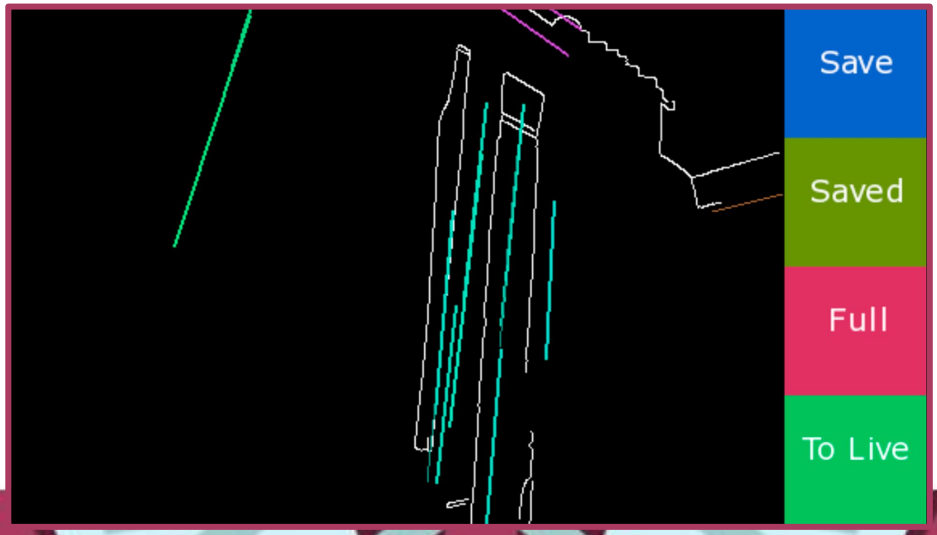
[1] https://archive.org/details/WarningForever
[2] https://en.wikipedia.org/wiki/ABA_Games

Seeds from the World

By Ethan Edwards
@winterblooms | www.ethanedwards.org

Procedural Generation algorithms often seem like magic. With a small uninteresting seed (a long number, some options in a start menu, the current time in milliseconds) this arcane set of instructions hidden behind layers of code can generate whole worlds, full of landscape, characters, music, and life. The algorithm takes something mundane and makes something interesting out of it. But what if you start from a seed that's already worth looking at?

Much of my recent work has tried to explore this question, generating music out of natural phenomena which in the first place rivals the artistic quality of what's produced. Rather than using a simple number to start, they use entities such as the present weather conditions or an artfully framed photograph taken by the user. The idea is not that a "better" seed might produce somehow better output, but that the algorithmic process of PCG might be able to allow for greater artistic appreciation of the seed itself.



Fox example, an iOS app I developed called Edge of the Sphere directly accesses the user's camera and displays back a live version of whatever they are seeing run through an edge detector. Suddenly the world of so many shapes and colors is reduced down to an abstracted monochrome of lines intersecting at various angles, only the outlines of the objects at hand. When the user taps the screen as in any normal photography app, these lines light up and produce synthesized sounds of many different pitches and durations. After using the app for a little while, the user beings to understand some parts of how it's done. Right angles produce pleasant sounding harmonies, grids (bookshelves, bricks) easily picked up by the camera produce many sounds. From here, one explores the possibility space through framing and the shapes of objects.

"Suddenly the world of so many shapes and colors is reduced down to an abstracted monochrome of lines intersecting at various angles, only the outlines of the objects at hand."

Ultimately the music is nothing groundbreaking. It is easy to produce, and played in a concert hall would sound like random noise. But what I hope to have done is create a direct association between the visuals and the sound which helps increase the appreciation for that original seed. My ideal case for this is that after playing around for a few minutes, the user will look around their so familiar room and begin to notice the angles and lines which have faced them for years but have gone unnoticed. Maybe on their walk to work the next day they'll spot a particularly shapely sewer grate or find a new angle to look at a bridge. The sounds are fun and interactive, but in the end they serve as bait back into the seed that generated them. The music must be just interesting enough that people keep thinking about the system and try to figure out how the generation works. Once they learn to use the lines to generate music, they better appreciate the geometry of our world which was already present.

The precise details of the algorithm really don't matter, only that the relationship between the input and output is understandable by humans. Angles determine the pitches and certain

geometrically nice angles generate pleasant sounds under the standard Western harmonic system. But any mapping would have worked, as long as the user can figure out for themselves that certain kinds of patterns, lengths, photographic framing, etc., can predictably produce results based off of those criteria. If similar seeds produce wildly different results, there is no real point to thinking about the seed and one might as well use a random number generator. The algorithm in this case is fully at the service of the input.

By no means am I the first or best do this sort of work. As algorithmic art has propagated, the desire for complex inputs has been satisfied many times. The artist Hannah Davis has created a system called TransProse which takes in texts such as novels or speeches and using sentiment analysis creates pieces of music, with the resulting pieces providing aural information about the source material. The idea of an output which sheds light on the source material is the very premise of data visualization, and hopefully as more artists approach this field we can get more exciting analysis of more mundane sources.

"The world contains beauty all around us, but under our current living conditions, we miss so much of it."

The world contains beauty all around us, but under our current living conditions, we miss so much of it. We scarcely notice the beauty of birdsong, the perfect geometry of cities, or the pleasure of a windy fall day in just the right light. Art can help rectify this situation by changing our perspective and preparing us to go out into the world and notice these things. As technology provides more and more tools for engaging people interactively and using data from our environment, endless possibilities emerge and it is up to artists to go out and find them.

Fairy Tales by a Computer

By Rick Hoppmann
@tinyruin | www.tinyworlds.org

Read how I made a program generating fairy tales.

"Before you were born there lived an intelligent musician. One day the musician was abducted by a barbarian. She tricked the barbarian. Then the musician fled in terror."

GENERATING THE PLOT

Studying the fairy tales of Brothers Grimm, I realized that a group of them worked after a similar structure.

You had one main hero, an evil opponent and a central conflict between them two. The story ends with a solution of the problem and sometimes also in a catastrophe.

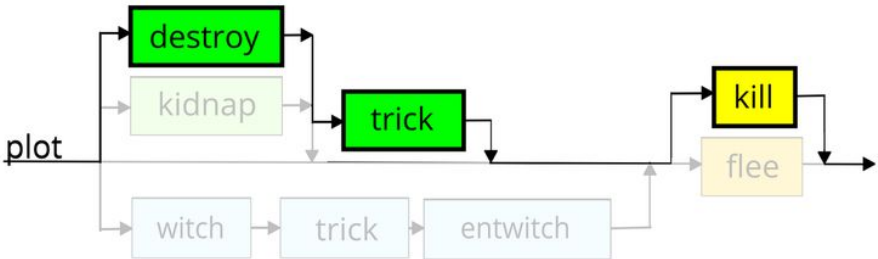
I started to go trough stories to gather ideas for common conflicts.

1. the evil actor kidnaps the hero or someone close to him

2. the evil actor bewitches the hero

3. the evil actor destroys a belonging of the hero

In order to generate the rest of the plot I now searched for a way to include the solution of the central conflict. The graph below illustrates how the plots are generated.



In this example the evil opponent destroys something of the hero. Then the hero tricks him. In the end the hero kills the evil opponent.

Now I had the core part of the programm done. The programm now knew how to generate plots for fairy tales. The next step was to teach it how to turn this plot into sentences and add some unique details to it.

TELLING THE STORY

At first I created sentence structures the program could choose out of. Some parts of the sentences are variable. They get a random word or wordgroup assigned out of a list. For example the destroy part is generated like this:

"There came a night" "a rascal" "demolished" "the hut" "of the archer."

Now the programm could also have choosen other words out of the list, making it turn into this:

"One day" "a rascal" "ruined" "the garden" "of the archer."

As you may have noticed, the hero and the evil opponent stay the same troughout the story.

The program decides at the beginning how to describe them both. Maybe the hero is a tiny mouse fighting an evil knight or the hero is a powerful mage fighting a strong barbarian.

With the sentence structures and the word lists added, the programm now knew how to narrate it's own fairy tales.

The Monster Deep in the woods there lived a wealthy hero. One day the hero was entranced by a monster.
The hero fooled the monster.
The monster ended the enchantment of the hero.
Thereupon she fled.

The Fighter In a tiny cave there lived a tiny rat.
One day the rat was carried off by a fighter.
The rat fooled the fighter.
After that she chained the fighter.

TRY IT YOURSELF

Download: www.bit.ly/plotnarrator (Windows, Linux) The source code is available under MIT license on github (https://github.com/tinyworlds/PlotNarrator).

Procedural Augmented Reality Game Content Generation Based on Players' Location

By Munir Makhmutov and Andrei Gusev

This work is dedicated to content generation problems in augmented reality games on mobile devices. Recently, the problems associated with Pokemon Go playing in unintended places have been widely publicized. In fact, there are a lot places where game content should not be generated because of ethical, religious and other reasons. This work shows a possible solution to the problem of game content generation in unauthorized places using a simple game example. To achieve this goal, it was decided to develop an algorithm which can check the whether generated content will be placed in prohibited locations. The results show that the algorithm works with most places, using clearly defined names of buildings taken from the Google Places API. This algorithm can be used in games with augmented reality based on location of players and any other applications working with objects on maps.

"This work shows a possible solution to the problem of game content generation in unauthorized places using a simple game example."

Augmented reality games are not new today. But even most popular augmented reality games based on maps like Pokemon Go do not provide the possibility to exclude prohibited locations such as churches, prisons, private ownership, etc. Therefore, game content (Pokemon in case of Pokemon Go) can be generated in and around prohibited areas. And due to this some of the players can be arrested for playing in these places. Other countries can just block such games on their territories entirely [1]. Initially, an analysis was conducted during which a list of the most unsuitable places for games was obtained. The list of forbidden locations is shown down below:

- Unclassified military installations
- Railroad tracks, stations and platforms
- Temples, mosques, synagogues and other religious places
- On the territories of state power objects
- In the cemeteries
- Hospitals
- Police stations

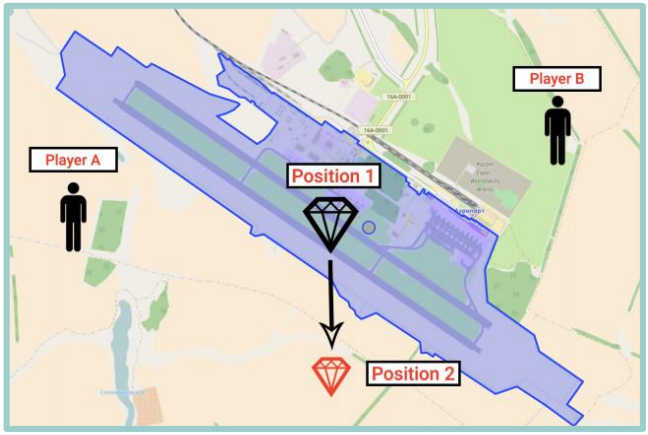
- Prisons
- On the border of the state
- Museums
- Airports and take-off stripes
- Water, erupting volcanoes and other dangerous places

Each of these places refers either to the private or state segment, or to places of spiritual value, playing in which can offend someone's feelings or rights. Generation of game content in water and volcanoes can endanger lives. A well-known story in the Washington Post [2], related to the Pokemon Go and The US Holocaust Memorial Museum shows importance of this study. "It's unacceptable to play Pokemon Go in a museum that is a memorial to the victims of Nazism," Andrew Hollinger, director of public affairs at the Holocaust museum, told CNN. After this incident, Niantic had to act and exclude some of museums from possible places of Pokemon appearance. But the problem did not disappear and the game remains blocked in some countries. And, of course, it is impossible to manually exclude all prohibited places. And to solve this problem, it was decided to create an algorithm that automates the exclusion of unwanted places. The list of forbidden locations was created for the most general cases, and it can be expanded in future. Some countries can have its own unique forbidden places.

"And to solve this problem, it was decided to create an algorithm that automates the exclusion of unwanted places."

A list of place types can be sent to the Google Places API to get all places of these types in a city or in radius from a particular place. But the problem is that Google returns only the information about the coordinates of a place, and even this is not necessarily the center of the object. Without place size, it is difficult to verify whether something is within a location or not. To solve this problem, OpenStreetMap API was used because it provides more detailed information about object location based on its name or coordinates. The query returns a set of points consisting of

latitude and longitude forming a complete outline of the area or building. These points form a polygon, belonging to which we should check. So the generation algorithm should find all prohibited areas using request to Google Places API, then a request containing these coordinates should be sent to OpenStreetMap API to get polygons. Based on these prohibited polygons and the alleged place of game content it should be decided if it possible to generate it or not. Generally, it does not matter which algorithm is used for game content generation. In case of our game the most important input data for algorithm is the set of players' locations. It is needed to generate content close enough to players. Figure 1 demonstrates generation of the content for Player A and Player B between which airport (prohibited area). If it is needed to generate one item for both of them, then a naieve solution would suggest the game content should be generated somewhere between them (Position 1), because it will be close enough to each other. However, this should not be done. Therefore, the algorithm should choose another position somewhere nearby, which could be Position 2 (depending on the implementation of the game).



"Figure 1. Game content generation near to prohibited area"

References

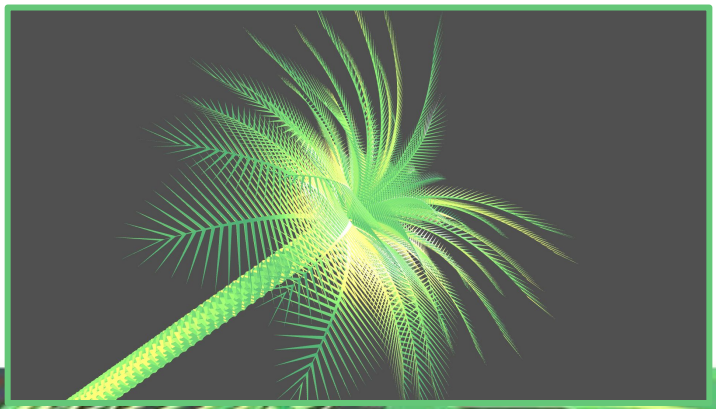
[1] S. Mans, "Who owns the playground?: Urban gamification and spatial politics in Pokemon GO," 2016. [Online]. Available: http://tinyurl.com/playwho

[2] A. Peterson, "Holocaust museum to visitors: Please stop catching Pokemon here," 12 July 2016. [Online]. Available: http://tinyurl.com/washpokego

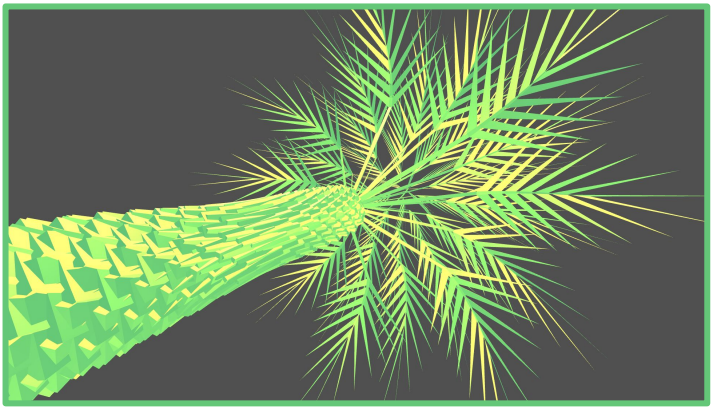
The Palm Generator

By Davide Prati
@edapx

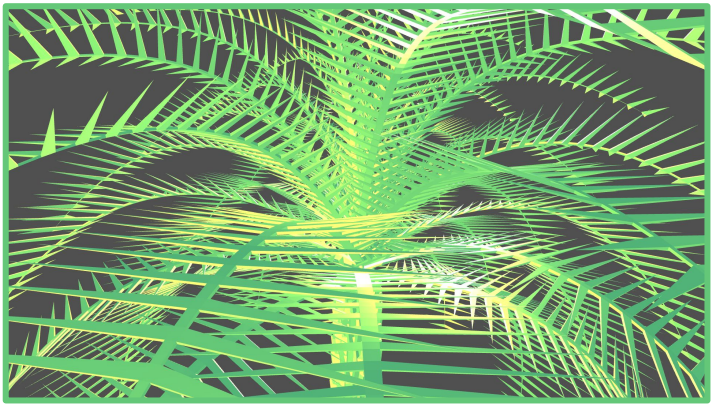
I did a Palm generator a while ago and I think it could be interesting for you. You can find all the information here.
http://davideprati.com/projects/palm-generator

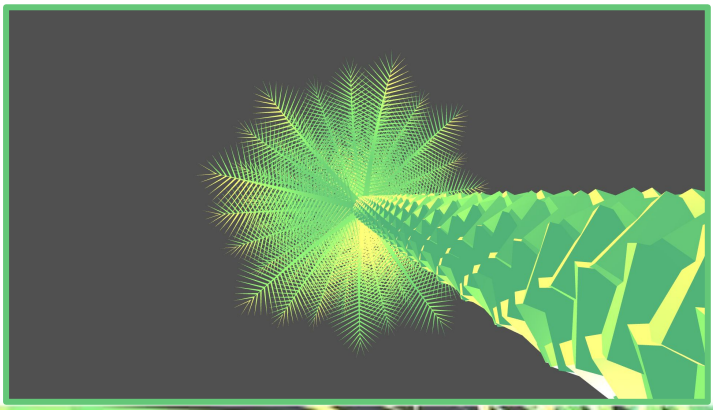


A zip with high res images can be downloaded here
https://drive.google.com/file/d/0B6sqjMvJRGMtU0FGQndCcUd6cUk/view



A poster with some palms it is available here http://davideprati.com/assets/media/palm-generator/palm-generator-poster.png





FIRKANT

By Peter 'tehwave' Jørgensen
@tehwave

FIRKANT is a fast-paced, procedural platformer set in a minimalistic, but expressive environment in which the player races against the decay of platforms to get the highest possible score. It's been in and out of development for Android & iOS since late 2015. Its key influences are Super Meat Boy and Cloudberry Kingdom.

Gameplay

The player controls a white square, which can be customized further to the player's liking, they can even draw their own sprite, to navigate endlessly through a procedurally generated set of platforms. Each platform has a limited lifespan, and will begin to fade out once it has spawned. The next platform will spawn when the player has collected a 'Generator' item present on the latest platform. This forces the player to be fast and keep moving, or else risk falling off the platform. For every item that the player collects, their score increases. The platforms vary in style. A portion of them will have spikes, while others will have springs, and some will even have enemies. As the player progresses through the endless level, the platforms that appear are increasingly more difficult to navigate.

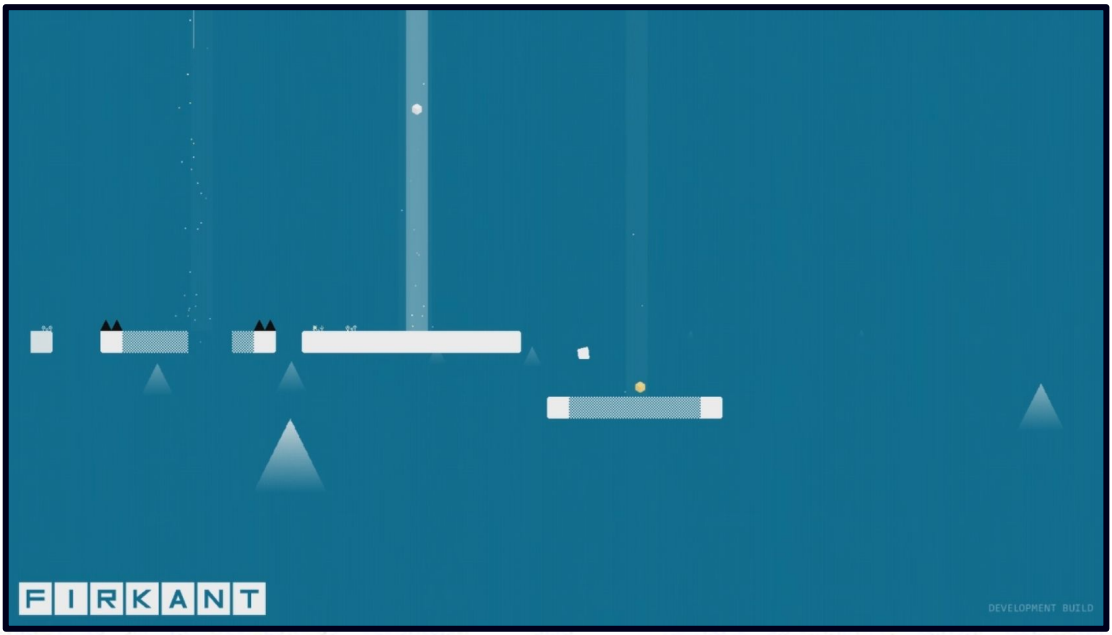
"This forces the player to be fast and keep moving, or else risk falling off the platform."

Procedural Generation

FIRKANT uses procedural generation to have the player stay invested, as they're expected to keep playing repeatedly to beat their high score. FIRKANT tracks how the player is playing by watching the time they spend on each platform, what they most often die to, how they're navigating, etc., to then adjust

the platforms accordingly to keep it just challenging enough for the player, so that they're not put off by the difficulty, and they want to keep trying to beat their high score. Each platform is designed to be increased in difficulty by adding hazards, springs, temporary platforms, and enemies to the platforms. Beyond this, platforms can have a set of rules applied to them, such as 'don't generate this if last platform was of this design,' which helps free up limitations to their design.

FIRKANT is the perfect candidate on how to effectively use procedural generation to keep gameplay exciting and fresh. For more information about FIRKANT, visit the website at https://peterchrjoergensen.dk/FIRKANT and subscribe to the newsletter via the form on the bottom of the page. You can follow its development on Twitter via @tehwave and #FIRKANT.



Affective Music Composition with MetaCompose

By Marco Scirea
@MarcoScirea | http://marcoscirea.com/

The game "The Witcher 3" (CD Projekt RED, 2015), a fantasy role playing game based on the novels by Polish author Andrzej Sapkowski, has been hailed as one of the best games of the last few years. The game has a very long and complex story which puts the player in front of non-trivial choices with long lasting consequences. While the music produced for the game is also of very high quality, as I was 10 to 20 hours in the game, my immersion started getting broken by the same tracks being repeated over and over again. Is it possible that in every inn the same music is always played?

Computer games have properties that make them particularly interesting and challenging for this kind of music generation: unlike traditional sequential media, such as novels or movies, events unfold in response to player input rather than a linear narrative. Therefore, a music composer for an interactive environment needs to create music that is dynamic, while also holding the listener's interest and avoiding repetition. This applies to a wide range of games although not all; for example rhythm games (e.g. Guitar Hero) make use of semi-static music around which the game-play is constructed.

"Therefore, a music composer for an interactive environment needs to create music that is dynamic, while also holding the listener's interest and avoiding repetition."

Moreover, creating dynamic music is not enough: music can be seen as a communication tool and we believe that it is important to use it to convey meaning. This research explores how to express narrative events through mood expressive music. It is important to note that in this domain there is not only a predefined narrative, but also an emergent one dictated by the player's actions.

The MetaCompose system was designed to create structured and varied music in real-time, through a combination of a graph

traversal-based chord sequence generator, an evolutionary search-based melody generator, a pattern-based accompaniment generator, and affective improvisation.

MetaCompose achieves this mood-expressive music by creating a composition (a chord sequence, a melody, a basic rhythm, and basic arpeggio), which is then rendered in real-time by a number of improvisational modules. These modules use a mood-expression theory based on a number of musical features (intensity, tempo, rhythm, timbre, and dissonances) to change the way they express the information contained in the composition, achieving different affective expression.

"How do players react to exposition to such music while playing a game?"

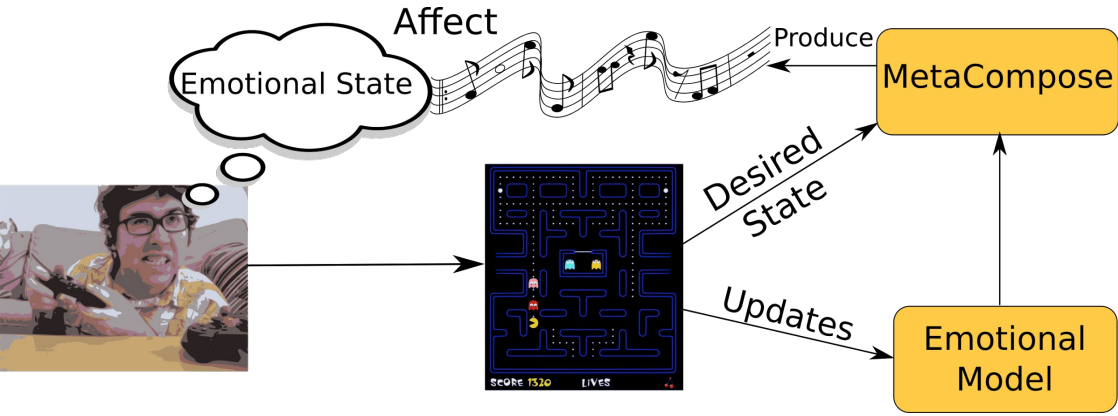
We have conducted several studies to validate the system's structure and its expressive capabilities. The results of our last study highlight how, while our mood expression theory is far from perfect, it allows for reliable expression of arousal and of strong differences in valence. Moreover transitions between affective states are generally correctly perceived which means that, while there might be disagreement between how to annotate the expression of a particular piece, participants distinguish changes in expression (e.g. from less to more negative).

But the real question that we should ask is: How do players react to exposition to such music while playing a game? We performed an experiment where MetaCompose has been used to create background music for the game of Checkers. Results show how the usage of the MetaCompose system while trying to express affect consistently with the current game state is better perceived by the players compared with random affective states and static expression.

Needless to say there are many improvements that the music

generation system would benefit from, and there are limitations in the domain itself. The most important of the latter are that the system is based on (and evaluated on) Western music, so its generality with different cultures is unclear, and that there is a strong difference between perceiving a mood and **feeling** a mood.

The most exciting extension of this work can be summarized in the figure.

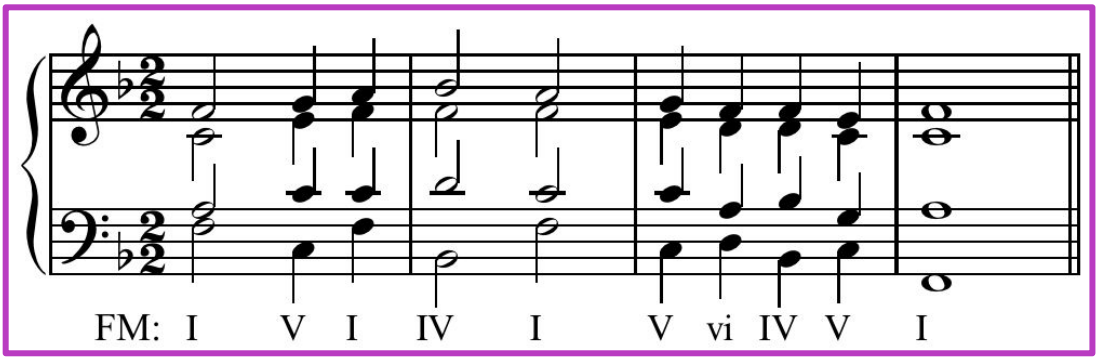


The player, being human, possesses an emotional state which might vary depending on the game. Of course this emotional state depends on many external factors besides the game but, as these factors are uncontrollable, when looking at this as a closed system we assume that changes are triggered by the interaction of the player with the other components of the system.
Players can get frustrated, find solace, relax, get excited, and much more by playing a game; what we would like to explore is the effect of affective music on this process, specifically if the use of specific affective expressive music could influence the player's experience and emotional state.

"Players can get frustrated, find solace, relax, get excited, and much more by playing a game..."

A clear problem arising from this system is how to detect the player's emotional state; an emotional model of the player could be created through data mining techniques on a combination of in-game data, self-report, and physiological measures.

This would allow us to experiment on how different expression in music might affect the player depending on her current state, ultimately creating more engaging and interesting experiences.
You can find out more about our findings and MetaCompose (and of course samples of music) on http://marcoscirea.com/.

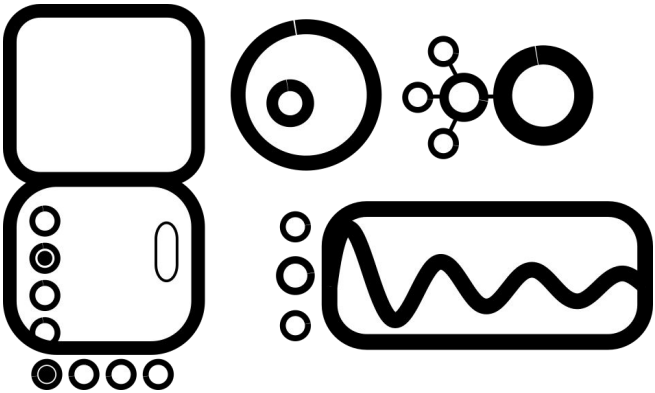


Shooting the Black Box

By Jasmine Otto
@jasminumlutris | https://github.com/jazztap

Here are interesting approaches toward helping you explore their game's possibility space. Or, here are some generative schemes that could only be pulled off with a human-in-the-loop.

Mu Cartographer (@Titouan_Millet)



In: 1 joystick + zoom, 1 sinusoidal wave decomposition, 1 graph, 1 box o' radio buttons

Out: view onto a dynamic topography, journal pages

None of the inputs are labelled for what they do, especially the ones which parameterize the topography. But conversion of joystick coordinates from polar to rectilinear are not much worse than the usual one-to-one mapping.

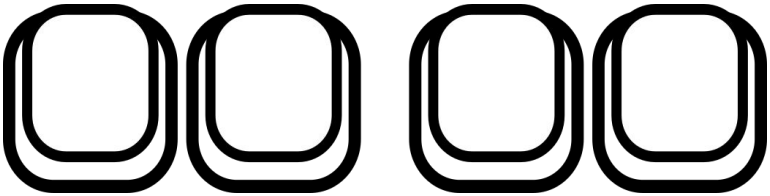
These are very generous black-box transformations, in the sense of having only a few unexpected behaviors, and eventually periodicity. Learning them reveals content, not just randomized minerals. Satisfying.

"But conversion of joystick coordinates from polar to rectilinear are not much worse than the usual one-to-one mapping."

Similar:

- GIMP's layer modes and brightness curves
- planted items which persist into your inventory, with attached story

QWOP (@bfod)



In: 4 buttons

Out: a Simulated Athlete

The inputs are labelled exactly for what they do. The vast majority of input sequences kick you out of the standing 'inverted pendulum' state, so you fall over. So unfortunately, this simple game is not nearly as learnable as our last one.

Alas, your horizontal speed is not mapped directly to two buttons, but instead emerges as you push yourself along (over) the ground with your legs. Biologically, this should really involve at least 8 inputs, for each hinge joint is stabilized by a pair of opposite muscles. At which point we may prefer to drive input from a 'central pattern generator', for example, a neural network with predefined weights and periodic output due to symmetry.

"But if the rhythm is even slightly out of sync, it would be wise to include a feedback which nudges our runner back into the unstable standing state, before it is too late."

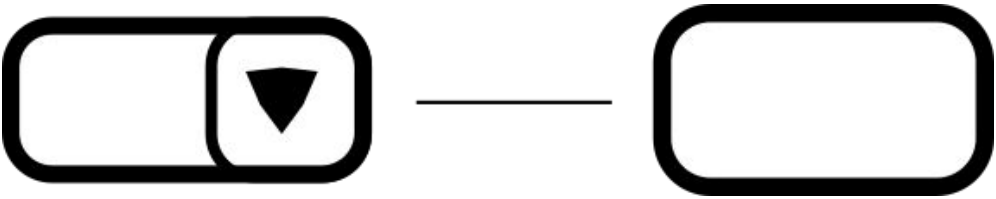
But if the rhythm is even slightly out of sync, it would be wise to include a feedback which nudges our runner back into the unstable standing state, before it is too late. I would love to know the relative weighting of this feedback, even though we both

know full well how to walk in any number of appropriate gaits, given only a direction and a speed. So the complexity abstracts itself away.

Similar:

- ‘How to animate cube in Houdini’
- Wii Tennis, played by ear

In a Wicked Age: four oracles (@lumpleygames)



In: 1 complexity dropdown, 1 reroll button

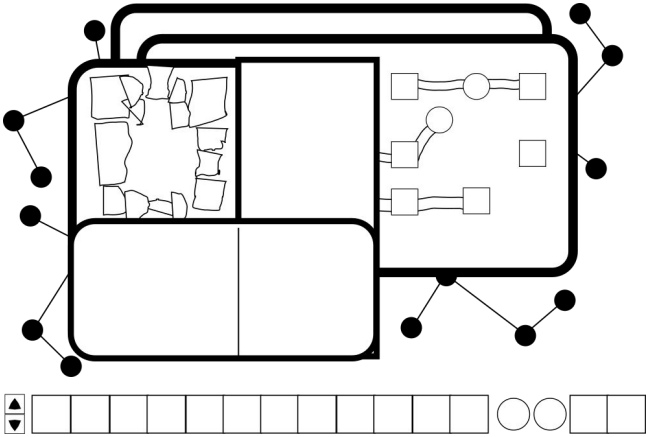
Out: assorted actors enmeshed in plots

This is a button that makes four stories. Well, it throws their pieces at you. You should probably steal more pieces when the one story you really like runs out of them? There are no rules.

Similar:

- WTF is my DND Character
- Fiasco

Grim Dawn (Crate Entertainment)



In: 12 hotbar buttons, 1 paper doll w/ inventory tetris + transfer area, 3 progression trees, 1 house points display, 1 journal, assorted shortcut buttons, 1 mouse

Out: a bad*ss pile of stats

In the narrow region between 'getting one-shot a lot' and 'press x to clear the screen', there is compelling incentive to learn your Most Spammable Ability, your Nuke, your Transient Buff(s), your Reposition, your Resummon, and your Panic Button(s).

Of course these will change in ten levels when a succession of sweet gear with stats viable for your class combo's *other* build drops, and you unlock an effect that makes your Nuke into a Resummon and gives your Reposition an AoE DoT.

There's also the usual cookie-clicker feeling, but shh, you're playing the keyboard *like a star*.

"There's also the usual cookie-clicker feeling, but shh, you're playing the keyboard like a star."

Similar:

- Lego Star Wars, in local co-op with oneself
- Fighting games on button-masher difficulty

Hadean Lands (@zarfeblong)

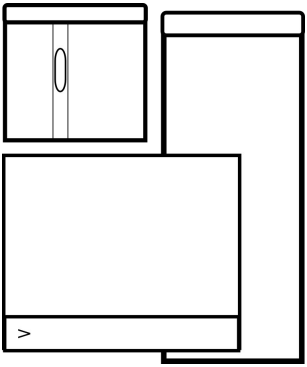
In: several highly context-sensitive verbs, 1 set of notes, 1 map

Out: prose

Sometimes the sequence of your input gets messed up, and you have to retype it all from the beginning. In some terminals you can press a sequence like (‘up’x4 ‘enter’)x3 to reproduce your last three actions before the reset exactly. In others you can declare rituals much longer than three lines explicitly, and rerun them from the top as need be.

In this parser, the higher-level procedures write themselves. Not only that, but they establish all of their pre-conditions. Not only that, but some pre-conditions are mutually exclusive (especially due to having one copy of a given ingredient), so you can re-write the ritual on the fly to not interfere with your next move.

It is absolutely freeing to mess around without having to re-do the last fifteen (ninety+) minutes of your life, and not even have to wait for the savegame.



Similar:

- Pyke, Prolog, and logic programming
- Jupyter, Sweave, and IRL reproducible research

Procedural Generation With Signed Distance Fields

By George Baron
@PixelbearGames | https://pixelbear.itch.io/

About Me

I am currently studying computer science as a masters student at the University of Sheffield, with a keen interest in graphics programming, art and game design. Recently as a hobby I have started to create demos akin to things you would find on the demoscene, and the following article is a result of my poking around!

"...different approach to procedural generation that instead renders a 3D world based purely on the mathematical definitions of objects.."

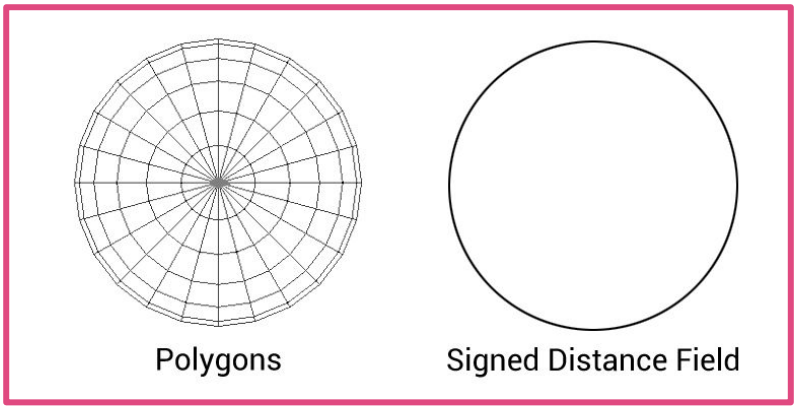
Introduction

Procedural generation comes in a wide variety of flavours, from using it to create believable open world environments to constructing little bits of dialogue for some random NPC. Most techniques rely on some kind of "library" of pre-defined outcomes to choose from; for instance, a procedurally generated world might place pre-designed assets such as trees and rocks into the world based on an algorithm.

What I am going to talk about in this article is an entirely different approach to procedural generation that instead renders a 3D world based purely on the mathematical definitions of objects. The technique is common practice over on Shadertoy, so if you are interested at all, Shadertoy is the perfect place to start!

What are Signed Distance Fields?

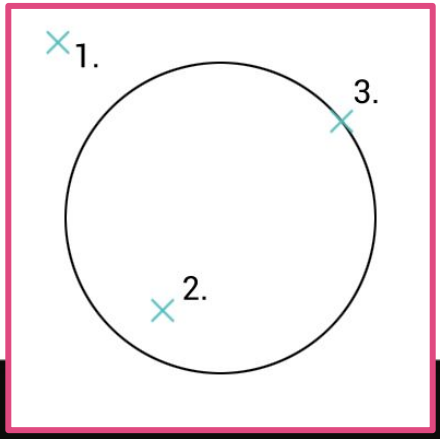
Signed distance fields are an entirely different approach to rendering. Instead of defining an object using a discrete number of triangles as you would in a typical application, signed distance fields represent the object with a purely mathematical definition, allowing for the continuous sampling of points. Below is a small diagram showing how a sphere looks with traditional polygonal-based 3D rendering, and how a sphere looks using signed distance fields!



The main thing to take away from the diagram is how smooth the edge of the signed distance field is compared to the polygonal sphere! So how exactly are signed distance fields defined? Quite simply, in fact! As the name suggests, they are based on "signed distance" - for signed distance fields, this means that for any point in space, a distance from the point to the edge of the field is taken. The distance can describe three different outcomes:

- Distance is positive, so the point is outside of the field.
- Distance is zero, so the point is on the edge of the field.
- Distance is negative, so the point is inside of the field.

The following diagram shows this more clearly.

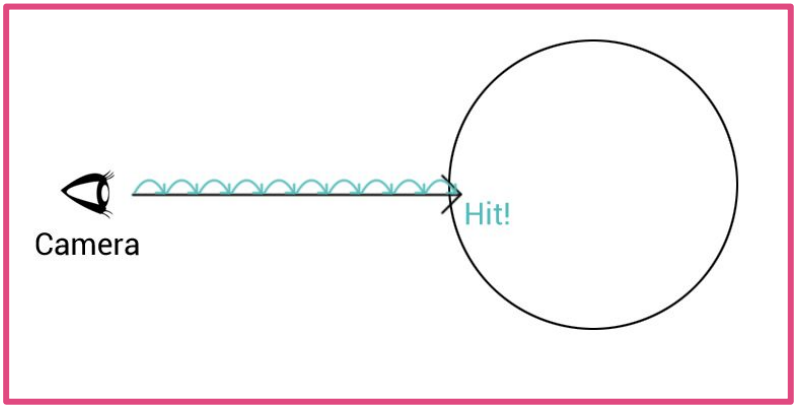


1. The point is outside of the field, so returns a positive distance from the point to the edge of the field.
2. The point is inside of the field, so returns a negative distance.
3. The point is on the edge of the field so returns zero. In most cases this would be taken as inside of the field.

By describing an object in this fashion, thousands of points could be randomly selected in space and suddenly an image of the object would start to become apparent. The only problem with this is that you can't just sample thousands of random points - we need to be selective to efficiently render the object!

Ray Marching

Ray marching is a rendering technique where, from a point in space, a ray is cast out in a particular direction and stepped or "marched" along incrementally. At each step, an approximate distance between the point on the ray and the signed distance field can be measured. If the distance returns negative at any of the steps, then the ray has collided with an object!



This technique can be used to render an entire scene. If you imagine the computer screen as a "window" into a 3D scene and you sat in front of the screen as a camera, a ray can be marched out from you, through a pixel in the screen and out into the 3D scene. By doing this

for every pixel in the screen, a 3D environment can be deduced and drawn efficiently.

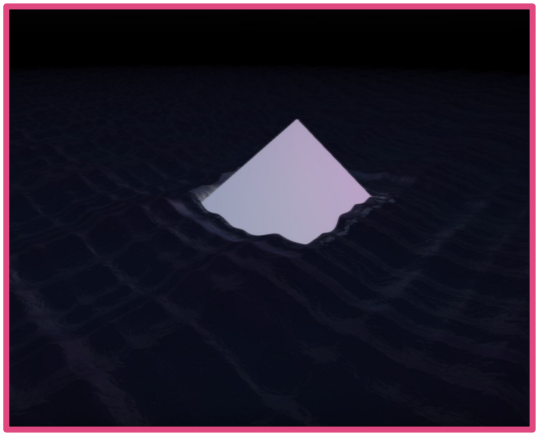
Shadertoy

Unfortunately I do not have enough time to detail an actual algorithm for this, but fear not! A really awesome interactive tutorial by TekF is available on Shadertoy here - https://www.shadertoy.com/view/MdBfRK. Shadertoy is a wonderful website for graphics programming that lets users write pixel shader programs and see the output immediately. If you are interested in procedurally generated imagery, then Shadertoy is an incredibly valuable resource!

"Shadertoy is a wonderful website for graphics programming that lets users write pixel shader programs and see the output immediately.

My Creations

It would be remiss of me to not show off some of the things I have made using the techniques described above. My interest lies in creating 3D environments that respond and evolve to music in real-time, using ray marching and signed distance fields. Below are a couple of screenshots of things that I have worked on!



"Bad News"

By James Ryan, Ben Samuel, & Adam Summerville
https://www.badnewsgame.com/

Bad News is an award-winning installation piece that combines procedural generation and live improvisational acting into an emotionally charged 45-minute experience. It has been performed at venues including IndieCade, where it won the 2016 Audience Choice award, Slamdance, and the San Francisco Museum of Modern Art. The project has also been featured in Gamasutra, The Guardian, and Rolling Stone, who proclaimed, "This marvel of procedural performance can only be played by a lucky few, and that's a crying shame."

"Being thus occupied, the mortician is forced to delegate this important task to his newly appointed assistant: the player."

The game takes place in the summer of 1979, where an unidentified resident of a small American town has died alone at home. The county mortician is responsible for identifying the body and notifying the next of kin, but a matter in a different part of the county demands his presence. Being thus occupied, the mortician is forced to delegate this important task to his newly appointed assistant: the player.

To carry out the task, the player must navigate the town and converse with its residents in order to obtain three crucial pieces of information, each of which can only be discovered by knowing the preceding piece: the identity of the deceased, given only the person's physical appearance and home; the identity of the next of kin, given the identity of the deceased and an explicit notion of a next of kin (which

is provided); and the current location of the next of kin, given his or her identity and any other relevant information that the player has gathered. Finally, upon locating the next of kin, the player must notify that person of the death. Throughout, she should remain discreet, so as to respect the privacy of the family.

To begin, the player is led by a guide to a quiet room, where she sits on one side of a constructed model theatre; on a raised surface with black tablecloth lies a tablet computer, a notebook, and a pen. A live actor sits across from the player, hidden by the theatre's adjustable curtain; behind a permanent lower curtain, a hidden screen displays a special actor interface and a concealed microphone captures sound. Out of sight, a wizard listens to the audio feed; prior to starting, he generated the unique town that the playthrough will take place in (more on this below).

From here, the player proceeds by speaking commands aloud; the wizard executes these throughout gameplay by live coding modifications to the simulation in real time. The most common commands are moving about the town (in a direction, or to an address), viewing a residential or business directory, approaching a character to engage in conversation, but by incorporating a human back-end almost any command is possible. As the player navigates the town, her interface updates to describe her current location.

"Each Bad News town is procedurally generated using the Talk of the Town AI framework."

When a player approaches a town resident, the hidden actor interface updates to display details about that character's personality, life history, and subjective beliefs. After spending a few moments preparing for the role, the actor pulls back the curtain to play that character live. As the subject of conversation shifts between residents of the town, the wizard crucially updates the actor interface to display the character's beliefs about that particular resident.

Meanwhile, the wizard queries the simulation for narrative intrigue (again by live coding), which he can deliver to the actor directly through a live chat session (e.g., "you were in a love triangle with the next of kin"). In performing a given town resident, the actor must adhere to that character's generated personality, life history, and subjective beliefs.

Gameplay ends once the player notifies the next of kin of the death. A typical session lasts roughly 45 minutes, though the wizard and actor can coordinate in real time to control this.

Each *Bad News* town is procedurally generated using the Talk of the Town AI framework. Employing a method inspired by *Dwarf Fortress*'s world-generation procedure, each town is simulated from its founding in 1839 up to the summer of 1979, when gameplay takes place. Over the course of this simulation, hundreds of generated town residents live out their lives, embedding themselves in rich social networks and forming subjective (often false) beliefs about the town. This provides an abundance of narrative

material and dramatic intrigue—family feuds, love triangles, struggling businesses—that exceeds the capacities of a 45-minute playthrough. A hand-authored experience might only be able to tractably fill out the lives of a dozen or so characters, but the procedural nature of *Bad News* means that every simulated person in the town has the same depth, allowing the player to go anywhere and talk to anyone without finding any seams.

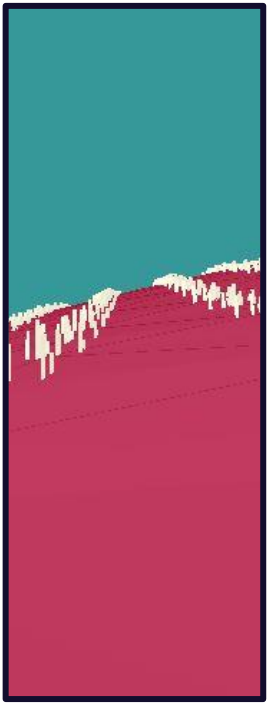
The exciting combination of technology and live improvisation is driving an emerging genre of related projects including Dietrich Squinkifer's *Coffee: A Misunderstanding* (2015), Twocan Consortium's *Séance* (2016), Ian Horswill's *Dear Leader's Happy Story Time* (2016), and Handsome Foxes in Vests's "The Truly Terrific Traveling Troubleshooter" (2017).

Bad News is project by Adam Summerville, Ben Samuel, and James Ryan. It is currently on hiatus.

Procedural Generation is Hard

By Owen
@newogame | http://owensoft.net/v4/item/2394

Despite what online video tutorials might lead you to believe procedural generation (procgen) in video games is hard, like really hard. You might as well skip height maps and go right to the hard stuff. The hard stuff will sneak up on you quickly like enemies popping into view in a n64 game. Coincidentally the hard stuff is the same point at which most tutorials reach 10 thousand lines of code and stuck in a corner to linger forever. I am not going to go down that path, all I am going to do is outline what I have learnt over the little time I spent working on my own little procgen side project.

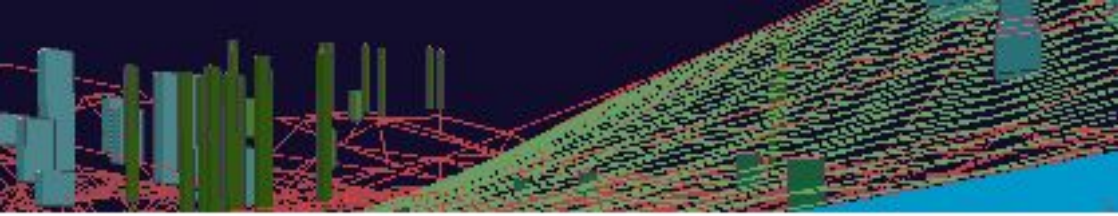


Randomness and Chaos

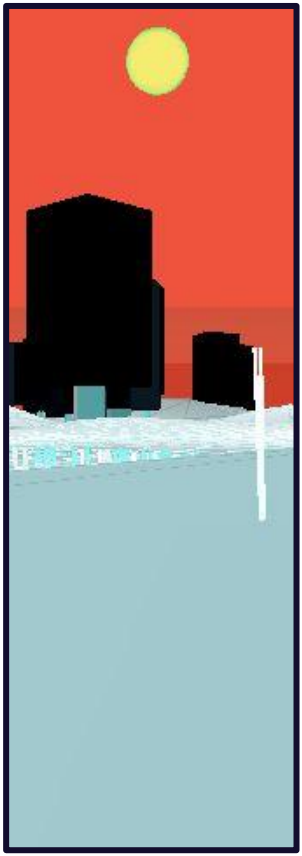
People seem to think of procedural generation (pg) as generating random points for use in a rendering graphics or content but soon they will discover that it is not about the randomness at all. Randomness is just a side effect that often leads to chaos when you are working with human beings. What you really need is constrained "predictability". Randomness is the least of your problems. Predictability is why so many tutorials use simplex/perlin noise. Simplex noise gives predictable results in a set range between -1..to..1 which you can use to produce predictable procedural content that only "seems" random. Randomness is NOT what you want. The goal is to have controlled space that appears random - not chaos.

Avoid simulating the real world

Some will attempt to create something like what was done in No Man's Sky or Minecraft but fail to understand that the real world does not work like a computer. No matter how much code you write or fast the code is you cannot simulate everything in the real world. And even if you try, it is going to take you a really, really, really long time just to get the basic stuff in place. You are gonna have to decide



what is important to simulate and what you will simply leave to the imagination. Game consoles and computers nowadays have upwards of 4 gigs of RAM and they still have to resort to trickery and shaders. It may seems like your favorite game is a true masterpiece but it is mostly smoke and mirrors once you figure out how it all works. This figuring out takes time - many dead bodies liter the highway.



Picking a noise function

A good noise function needs to be fast and is often complicated to code so it is best to stick with simplex noise or something similar. Tutorials often state that you can use any noise function you want but you want to avoid functions that are "random" as mentioned earlier - you want to know what result you are going to get. And you want to be able to repeat previous results so that you can test different outcomes in different scenarios.

Draw distance, view space, LODs, cache and culling

You will be tempted to fill your screen with tonnes of flowers and trees and such early on until you realize that your computer cannot keep up with all the information. The world is full of information, more information that you can imagine. Most pretty video games that you see being published today are using all sorts of trickery to render the information you see. It is not only down to the skill of the program but Art, RAM and disk read speed. The level of detail (LOD) of the environment has a lot to do with it as well. Some games have asset pipelines that auto-generate thousands of models just for one scene and then stream them in seamlessly. Other games store entire static sections of the game world on disk or in memory during loading instead of rendering it all in real time. You will have to sacrifice something depending on your situation. If you cannot pay 10,000

artists then you might want to keep your scope down to something that you can mange.

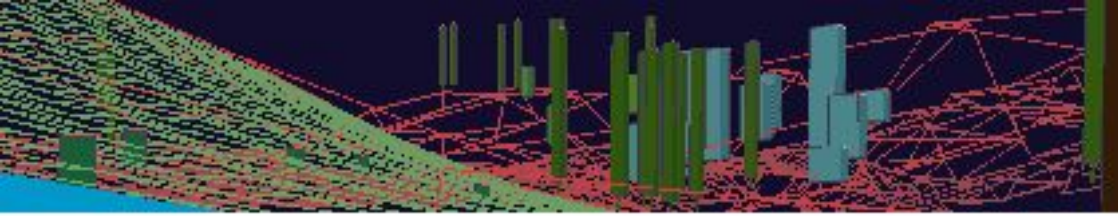
Distractions are everywhere

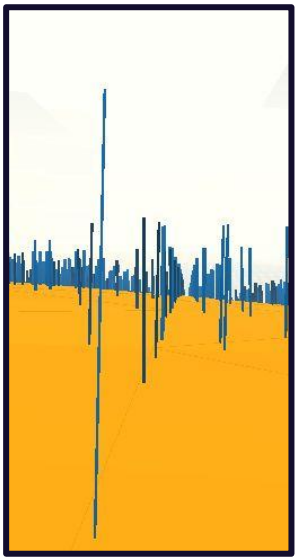
Feature/Scope creep is worse in procedural generation. Everybody wants everything as the case was in No Man's Sky. You think you can program every leaf on a plant blowing in the wind on a moon of Saturn but how much time do you have in the day? Not allot. And you will see someone using a cool shader in Unity and you will be like: "I want that in my code too!". Avoid the temptation. That stuff will only lead you down a road of pain and frustration. That fancy shader is probably using up 90% of the available CPU/GPU cycles - leaving nothing left for any kind of game play. It took me almost a year to get wave animations on bodies of water. It just came to mean one night but the key lesson is being able to notice when a "nice-to-have" feature is wasting you productive time and being able to leave it alone and move on to something more important.

"There might be objects that are longer needed such as dead enemies and trees that you have already cut down."

Keeping track of everything

It is good to be generating lots of stuff but you will come to a point where you need to know where something is or was and some kind of key/id to identify it. There might be objects that are longer needed such as dead enemies and trees that you have already cut down. The solution that I have is to keep track/save the xyz point at which I generated the object. In my system I try to ensure that only one thing is generated at every point in the world. So even if the object has moved to another spot I can check the object listing to see if something with an matching origin point has already been generated or if I need to generate it for the first time. This only applies to things that can be killed or have AI movement. Most things would otherwise not need to be tracked.



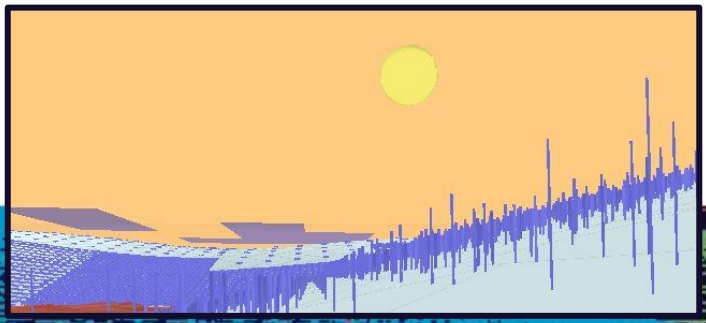


When the going gets tough

While writing your #procgen thing one of the first challenges you will come up against is how to place stuff in the world without them overlapping against each other, floating trees, planets colliding and caves - Oh god caves! This is what a lot of online tutorials fail to tell you: if you start by using height maps you are going run into a wall where you cannot do caves and you end up putting too much data into cache. Loops, in loops, in loops, then you hit the point when you try to solve it with threading - anytime you see someone start using threading you know that it is going to be a roller coaster down the rabbit hole. Avoid threading unless it is the last thing you implement - its a rabbit hole - it will not help. If the code is too slow to run on the main thread at such an early stage then you are doing too much. Stop - hammer time.

Conclusion

In the end procgen is not about replay-ability or creating lots of content or infinite worlds or pretty graphics or whatever people are hyping nowadays. Procgen is (at its root) about maintaining state of a complex system. You can generate a billion pieces of content but if you cannot put them together in a cohesive system then you are basically just generating garbage at which point you might as well hand make the level and be done with it. You can only play a game that you finish making. Expect to spend a great deal of time figuring out how to keep the state of all the stuff you are generating. Anyway good luck and happy proc-gen-ing!



By the Light of the Sun We Go

By Jason Grinblat

How the Twitter Bot *Unknown Peoples* Evokes Such a Vividly Gentle World

The People of the Eastern Depths. They love to tell stories about the fall of the Bird City.

> What music do they make?

Their pipe players wear platinum-scale belts.

> What is their architecture?

Their architecture is based on the triangle.

> What do they eat?

They prefer to eat a simple bowl of shellfish and offal.

The exchange above does something magical for me. It paints an intimate portrait of a mysterious people while inviting me to puzzle out the meaning of their customs. Why is their homeland called the Eastern Depths? What does it mean for their architecture to be based on the triangle? What is the Bird City, and what happened there? Its brush strokes are broad but its colors are so vivid.

The exchange comes from a conversation between a Twitter user and an interactive bot called Unknown Peoples (@neighbour_civs). The bot, whose words are bolded, tweets descriptions of cultures from across a vast, pretend world. It's part ethnographic tome, part AI oracle, and part text adventure game. Over the course of its several thousand tweet lifespan, it's conjured into being a wondrous and tender world.

A follower's interpretation of an Unknown Peoples's tweet. (credit: James Head, @djemps)

Unknown Peoples was created by artist and game maker G.P. Lackey (@mousefountain)[1]. Beyond the usual questions that preoccupy the fantasy genre, G.P.'s work tends to ask questions like: what does comfort look like? How do people share? How do they celebrate? This kind of examination of a world's soft edges—this *otherworldly gentleness*—typifies *Unknown Peoples*'s output.

[1] With small contributions by Tanya X. Short (@tanyaxshort) and myself (@ptychomancer).

The bot was created with *Cheap Bots, Done Quick!*, a beautiful little technology stack that lets artists make Twitter bots without needing to know how to code. CBDQ's bots are stateless, meaning they have no memory of their former tweets. For *Unknown Peoples*, this means that each culture's description is generated in isolation. Even the replies to questions about a particular culture, like the ones above, are generated without knowledge of the original tweet. So how can the output feel so coherent? How does the sense of otherworldly gentless float from culture to culture, custom to custom, tweet to tweet? A novel's author might spend several chapters layering meaning and mood to produce this kind

of coherence. The bot maker lacks this affordance and must instead conjure it from disparate parts. They share some of the same tools of language, but even more fundamental to the bot maker is the act of proceduralization—the deconstruction of something like a culture into its constituent objects and relationships—and how that process encodes meaning.

Unknown Peoples, like all CBDQ bots, uses a tool called *Tracery* to define a generative grammar that produces its output. A grammar is simply a set of nested rules [2]. Grammars are used to generate all kinds of output, but text generation is one of their most common and straightforward uses. In writing the rules for *Unknown Peoples*, G.P. took his mental model for what it means to be a culture and chopped it up into a set of objects and relationships. There's a tremendous amount of expression in this process. Choosing how to represent something—enumerating the properties that comprise it, and conversely, omitting the ones that don't—is making an argument about how we should perceive it. *Unknown Peoples* encodes *culture-ness* as rules about food, art, music, sports, and fashion. In doing so, it constructs a vision for what a culture is, or at the very least how we might choose to see it.

[2] Check out Kate Compton's wonderful blog post on generators for details about grammars: http://galaxykate0.tumblr.com/post/139774965871/so-you-want-to-build-a-generator

```
"statementToplevelAll": [
    "#statementWayofLife#",
    "#statementBodyDescription#",
    "#statementFashionBase#",
    "#statementReligion#",
    "#statementMartial#",
    "#statementFashionAccessory#",
    "#statementSocial#",
    "#statementFood#",
    "#statementArt#",
    "#statementGreatBuilding#",
    "#statementFashionMakeup#",
    "#statementNomad#",
    "#statementFashionHairstyles#",
    "#statementHoliday#",
    "#statementArchitecture#",
    "#statementMyth#",
    "#statementSports#",
    "#statementOdd#",
    "#statementMusic#",
    "#statementTrade#",
    "#statementRelationship#",
    "#statementStory#",
    "#statementTime#",
    "#statementTechnology#",
    "#statementLeader#",
    "#statementCalendar#",
    "#statementShipDescription#"
],
```

The rule for choosing what type of statement composes a culture's top-level description. It encodes certain beliefs about what's culturally important, and in doing so it contributes to a perspective the bot is offering and an argument it's making.

These kinds of representational choices are made at all levels of the grammar. In writing the rule for *#statementFood#*, G.P. specifies what food looks like in this world and what cultural roles it might play.

```
"statementFood": [
    "Their national dish is #foodCombinedDish#.",
    "They love to partake of a #beverageCombined#.",
    "They take their meals of #foodIngredientPrime# with a #beverageCombined#.",
    "They are known for a particular kind of food: #foodCombinedDish#.",
    "Their people are very fond of #foodCombinedDish#.",
    "Their cooks often prepare #foodCombinedDish#.",
    "They often eat #foodCombinedDish#.",
    "Their cuisine relies heavily on #foodIngredientPrime#.",
    "Their cuisine is mostly based on #foodFlavourType# flavours.",
    "They are known for #foodQualifier# #foodIngredientPrime# dishes.",
    "They care only to eat #foodQualifier# #foodIngredientPrime#.",
    "They prefer to eat a simple bowl of #foodIngredientPrime# and #foodIngredientRare#.",
    "A delicacy of theirs is #foodCombinedDish#.",
    "They carefully prepare meals of #foodIngredientRare# and #foodIngredientPrime#.",
    "They subsist almost entirely on #foodIngredientPrime#.",
    "Once in a while they enjoy #foodSpecialTreat#.",
    "It is customary for them to share meals of #foodCombinedDish#.",
    "They delight in many varieties of #foodIngredientPrime#.",
    "They have a rapacious appetite for #foodIngredientRare#.",
    "None can match their thirst for #beverageCombined#.",
    "Every family makes their own #beverageCombined#.",
    "Each clan has a different recipe for #foodCombinedDish#.",
    "#statementFoodRare#"
],
```

The rule for describing a culture's food.

The same is true for all sorts of choices: what relationships look like, how trade is conducted, and what kinds of stories are shared, to name a few.

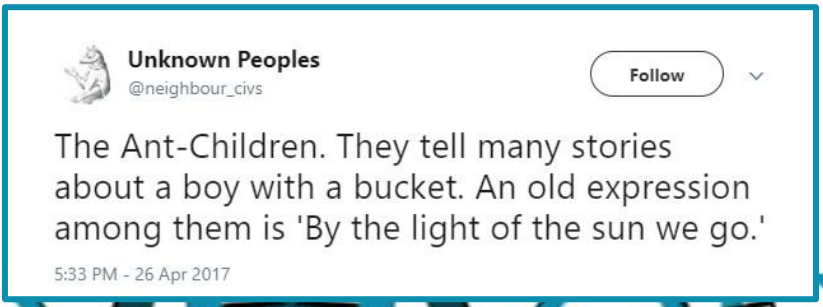
The terminal rules—those whose values don't contain new symbols—are often simple word lists that are consumed by the higher-level rules. Examples include meal ingredients, building materials, and nautical adjectives. Here the bot maker employs the familiar tool of diction. By choosing which words to include among the list of possibilities, the bot maker infuses their output with a certain tone and mood. G.P. has chosen naturalistic ingredients that paint a pastoral picture.

Food and beverage ingredient word lists. The diction gives the output a pastoral feel.

```
"foodIngredientPrime": [
    "fish",
    "eggs",
    "beans",
    "seeds",
    "nuts",
    "meat",
    "game meat",
    "poultry",
    "wildberry",
    "honey",
    "tuber",
    "fruit",
    "shellfish",
    "cheese",
    "mushroom",
    "rice",
    "seaweed",
    "grain",
    "vegetables",
    "#foodIngredientRare#"
],
```

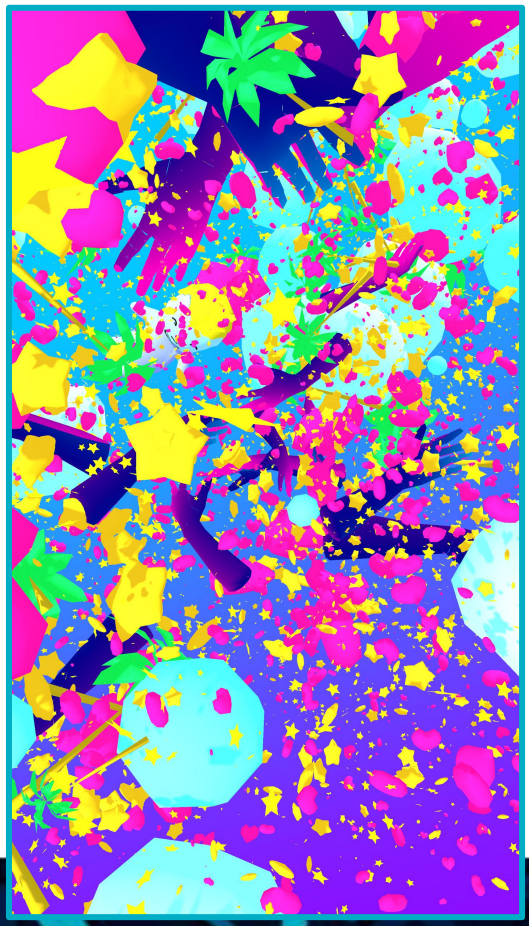
```
"ingredientBrewable": [
    "yam",
    "aloe",
    "marrow",
    "pepper",
    "yogurt",
    "honey",
    "corn",
    "reeds",
    "melon",
    "fruit",
    "berry",
    "cactus",
    "fungus"
],
```

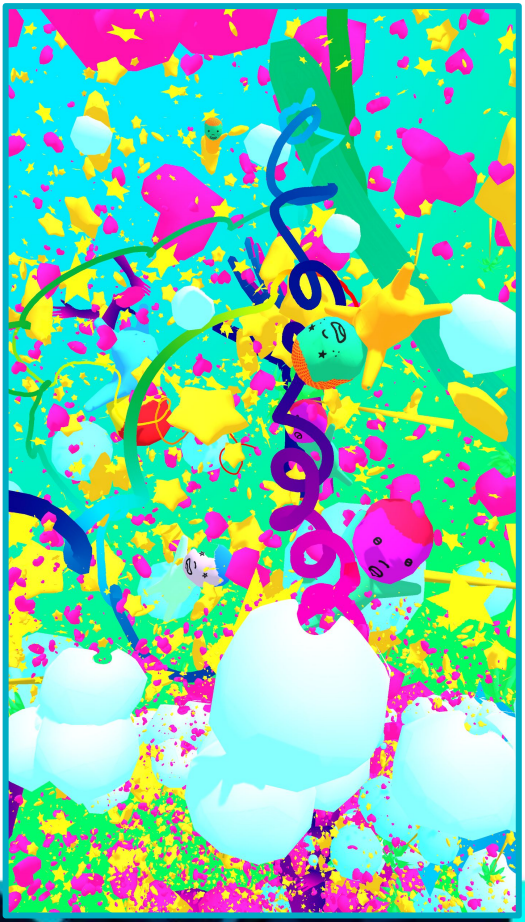
As a whole, *Unknown Peoples*'s grammar acts as a sort of proceduralization of G.P.'s voice as a creator. It encodes the perspective, mood, tone, and vision that he might manually assimilate into a wholly handcrafted project. But by virtue of *Unknown Peoples*' procedurality, we get to continually engage with new and surprising manifestations of this vivid, gentle world.

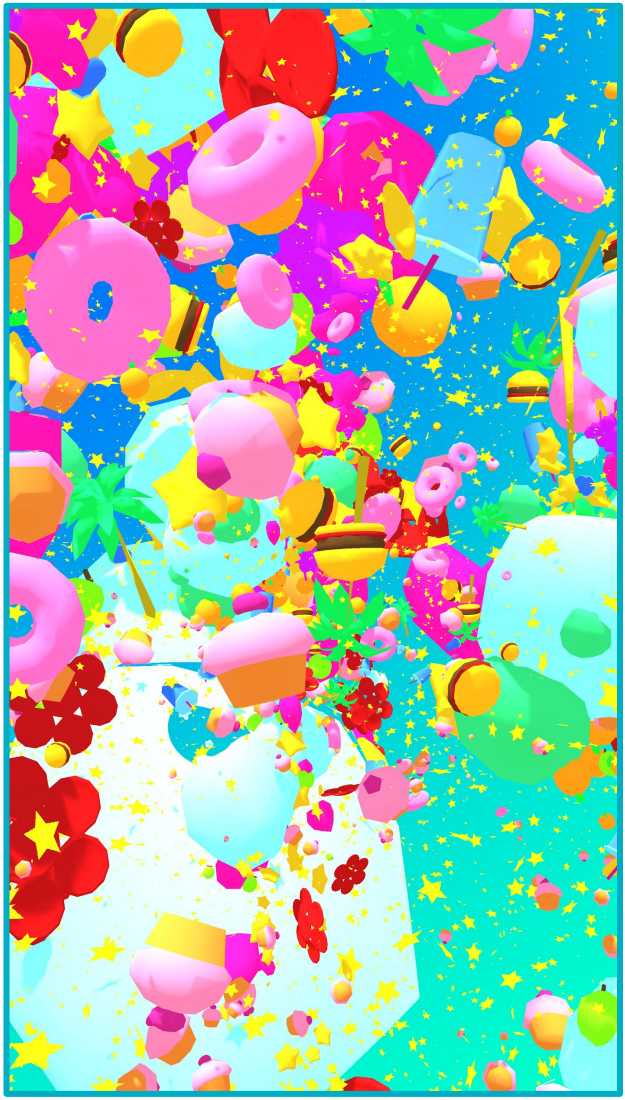


Hoopla Hoo

By Matthew Keff
@matthewkeff | http://matthewkeff.com/







This image was used for the cover of this Seeds Zine (Issue 2)

THE DINOSAUR GENERATOR

By Elle Sullivan
@THISISDINOSAUR

One thing that has always struck me about a lot of attempts at procedurally generating creatures is how many of them tend to fall at either end of a spectrum. On one end, they work by making changes to a base mesh, by changing textures, adding or removing pre built parts (like horns), or sometimes making more significant changes, like stretching or contracting limbs in No Mans Sky (take a drink), or the more detail oriented changes affording by things like the Sims character editor. This kind of approach can give highly predictable results, and the results can be much easier to work with (for example, animating an unchanging base mesh is relatively straight forward), however they can be quite limited, given the relative inflexibility of the original base mesh, and are unlikely to surprise you. and tender world.

This stands in stark contrast to the systems at the other end of the spectrum, that generate from practically nothing, usually using evolutionary methods (e.g. genetic algorithms); Karl Sims' creatures is possibly the most famous of this class of generators. This kind of approach can produce a wide variety of interesting things, potentially well beyond the imagination of the person that created the generator, but they can be almost impossible to work with; it's difficult to get them generating anything at all, and then it's even harder to work with the result: you need a system that can animate anything, and you have no information to inform anything else (for example, what kind of environment does your newly generated creature live in? You need some sort of selection criteria to create something that looks like it truly belongs).

To me, the most interesting kind of generators are therefore those that fall as far towards the latter end of this spectrum as possible, whilst still being fully capable of integrating its results into a project (e.g. a game) in interesting ways. Spore was particularly interesting

in this regard (take a drink), with its well constrained generation but with a large flexibility of body shapes, configurations of legs etc, but, like no man's sky (drink), the results are fairly alien, and there's a significant reliance on less flexible add on parts.

The evolutionary approach in particular I think is usually far too random and undirected, and too ambitious. They typically generate from nothing, trying to simulate millions of years of evolution. Say if we are trying to make a general animal generator, we have a lot of knowledge of animals and evolution, so I'm interested in leveraging that to give computers a head start. I am therefore interested in approaches that are more in the middle of the spectrum, where the generation is very directed, but starts from a more abstract and flexible point than a base mesh; the idea being that the end results are easy to predict and control, and that should afford all manner of interesting uses, thanks to meta data from the generation process (more on that later). It is with this in mind that I came up with The Dinosaur Generator.

"They typically generate from nothing, trying to simulate millions of years of evolution"

The original idea for this project came from the fact that Dinosauria isn't really that diverse (at least when compared with say, all mammals), consisting of theropods (carnivores), sauropods (the long necked dinosaurs), ornithopods (bipedal herbivores), and the various types of armoured dinosaurs. It should therefore be possible to use a domain specific knowledge based approach, with hand written constraints, to generate all the dinosaurs. So, the first step is to create an engine that can produce all the dinosaurs (and then some), and to then constrain it to only produce feasible dinosaurs (e.g. there's a relationship between the size of the neck, the general posture, and the size of the tail, otherwise the creature would fall over).

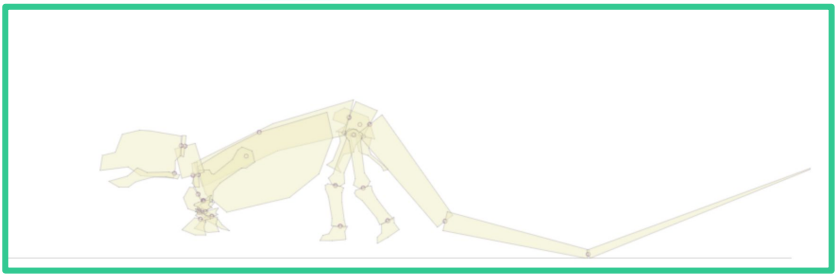


Figure One: an infeasible dinosaur, produced when the engine has few constraints (and a friend is given access to the sliders)

This lead to me producing a DSL (Domain Specific Language) to specify parameterised anatomies (e.g. a skeleton with variables, like tail length and scapula size) that supports interdependencies, so things can change not just based on the parameters directly, but can be functions of other parts of the anatomy, so the length of the tail could depend on not just on the tailLength parameter, but could be a complicated combination of also the neck length, the size of the head, the body angle etc. This can allow for highly constrained anatomies, or ones that are barely constrained at all (e.g. this could be written to be heavily parameterised and have no interrelationships or constraints (i.e. every bone could be at any angle, size, length etc.), or in theory this could be written to be so heavily manually constrained that no matter the parameters, every output is a feasible dinosaur, and that it's capable of producing every dinosaur).

Once you have such an anatomy description (constrained or otherwise), you can parameterise known dinosaurs in the system, which then allows you to do all kinds of cool things. With the real

Figure two: a small snippet of the language, defining two bones, and two connections.

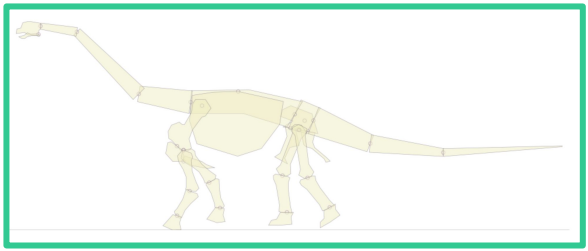
```
var: cervicalFirstLength = 210 + tailFirstLengthChange
cervicalFirst = trapesium(cervicalFirstLength, cervicalFirstLength,
cervicalFirstStartHeight, cervicalMiddleStartHeight)
cervicalMiddle~cervicalFirst = average(0, last) ~ average(1, 2), -neckAngle * 0.8
- bodyAngle * 0.5
//neck end

//head start
cranium = [140, 11], [140, 44], [130, 53], [130, 67], [127, 83], [71, 79], [49,
64], [1, 70], [5, 44], [27, 39], [39, 8], [69, 1], [86, 2]
cervicalFirst~cranium = average(0, last) ~ average(0, 1), -neckAngle * 0.2 -
bodyAngle * 0.5
```

world knowledge of existing dinosaurs, you could do similarity comparisons of any generated dinosaur to find its closest relatives and construct fictitious phylogenetic trees. You could then use this to inform name generation. You could even do things like use this to populate areas of a game where fossilisation is less likely (i.e. areas where we know less about the dinosaurs that inhabited them) with dinosaurs that are more different to any known dinosaurs, or to figure out where a generated dinosaur was most likely to live and to populate areas appropriately. You could also use it to inform its relationships with other dinosaurs (both generated and real), like predator prey relationships. You can use it to inform its diet, its mating habits, how likely it was to have feathers, how fast it is, it's most likely cause of death, and anything else you can think of. Now that we have these parameterised existing dinosaurs, this leads to the realisation that we could use them to probabilistically infer constraints, instead of trying to manually write them. (E.g. if we have a dinosaur with a massive head, long neck and tiny tail, there would be no dinosaur in the set of existing dinosaurs remotely close to this combination of parameters, so the probability of it being generated would be incredibly small). This could be done with a number of methods, such as using machine learning to create a machine that says whether any particular set of parameters is feasible or not (or gives a continues feasibility score), or perhaps a more straightforward method, such as selecting each parameter value as some function of how likely each specific value is to be in the set of existing dinosaurs (and each subsequent parameter could then be selected based on the likelihood of each value given the already selected values). If required, to bulk out the data it's not even necessary to use existing dinosaurs, the generator can be set to randomly generate large numbers of dinosaurs, and they can then be marked as feasible or infeasible (although, depending on constraints, there may be too many unfeasible to feasible ones, so existing dinosaurs could be used to bias the generation procedure for this

"You could also use it to inform its relationships with other dinosaurs (both generated and real), like predator prey relationships."

data could be used to bias the generation procedure for this data collection). So, with this approach, the generator is still domain specific, but the domain specific knowledge goes only into the creation of the generator engine, rather than in the manual writing of constraints on the output of that engine (although it's certainly still possible do both).

At present, The Dinosaur Generator is currently not really concerned with anatomical accuracy, as long as the dinosaurs look feasible. I believe the general approach isn't incompatible with anatomical accuracy and precision, but would certainly take a lot longer and require more complexity/detail in the anatomy specification.



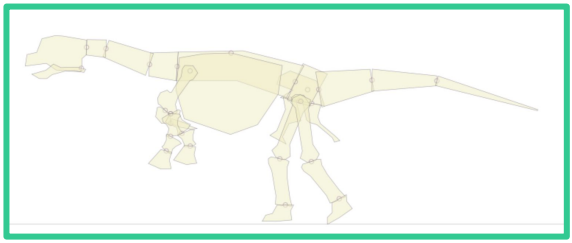


Figure three: Top, a sauropod produced by The Dinosaur Generator. Bottom, a theropod

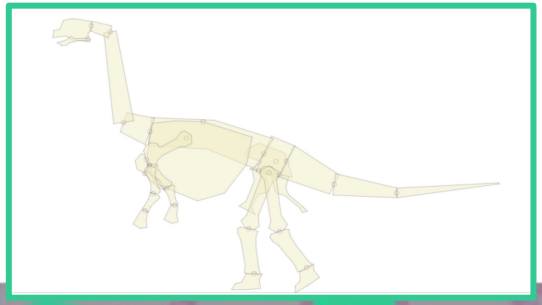
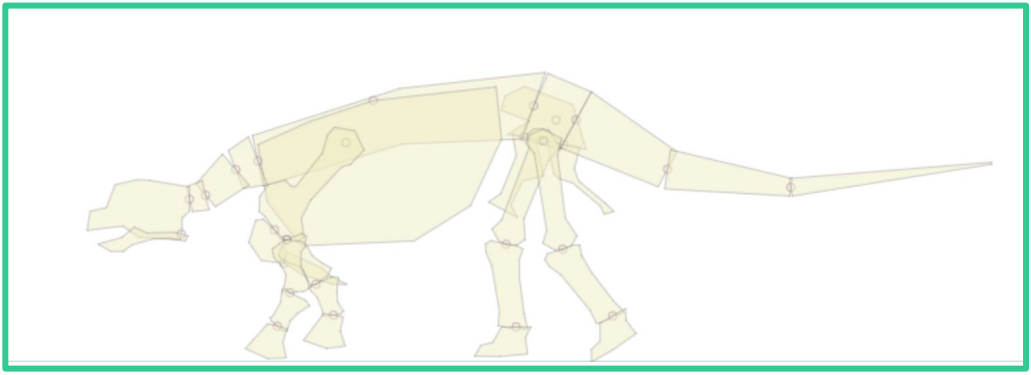


Figure four: An ornithomimosaur, an ostrich like theropod.

I think one of the most interesting things to come in this project is seeing how far it can be generalised for other creatures. Birds are an obvious first step, but the most interesting thing will be finding out how much manual constraining will be required as the scope is expanded, how much the complexity of the constraints required will grow with the scope, and just how wide the scope of a single anatomy description can be. For example, it may be possible to write a single very general and unconstrained specification that can generate birds, dinosaurs, and even mammals; or they each might have to be a separate specification, each with large numbers of hand coded rules to help limit it only to animals that are feasible. Another interesting avenue to explore is beyond real animals, to see if this approach can be used to generate alien creatures, whilst maintaining the benefits gained from the meta data.

Figure 5



Of course, one of the main challenges of an overall generation approach like Figure five: It is early days for the armoured dinosaurs Figure six: It is the relative similarity between seemingly disparate animals that hopefully means this approach will be capable of producing a wide range of different animals Image take from Evolution in Action (photographs by Patrick Gries) this is then in using the results; in the mesh generation, the rigging, the animation, and in the other steps required to fully

integrate the output into a project, but I believe these are all areas where the metadata will once again be invaluable. For example, generated dinosaurs could be animated by blending combinations of hand animated existing dinosaurs, again based on the similarity comparison done before. Still, there are no doubt significant technical and conceptual challenges ahead for the project if it is to get to a point where the assets could be used in a completely unconstrained 3D game.

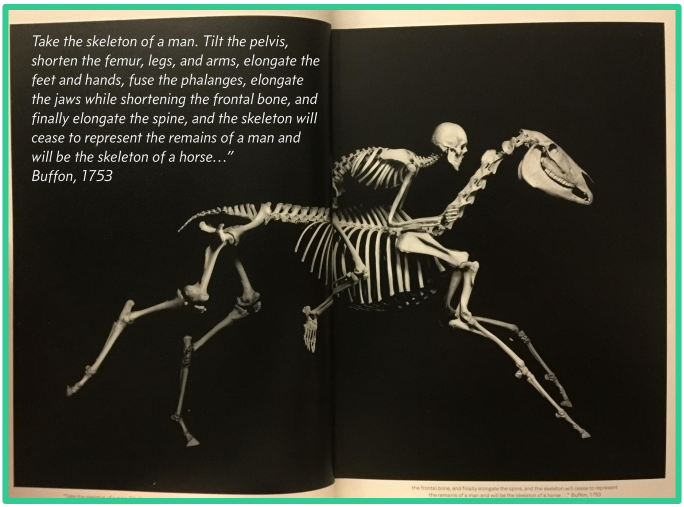


Figure 6

Tile Based Genetic Programming Generation for Diablo-like games

By Joseph Alexander Brown & Valtchan Valtchanov
@jb03hf

Diablo's initial pitch document highlights the use of Procedural Content as a prominent feature of the game:

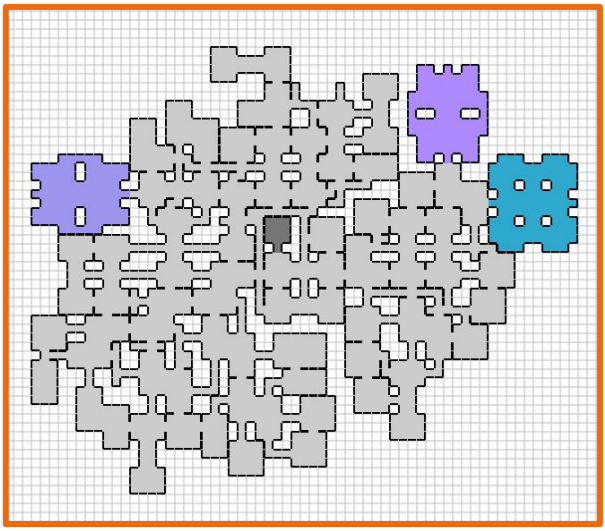
"The heart of Diablo is the randomly created dungeon. A new dungeon level is generated each time a level is entered, using our Dynamic Random Level Generator (DRLG) System. Rooms, corridors, traps, treasures, monsters and stairways will be randomly placed, providing a new gaming experience every time Diablo is played. In addition to the random halls and rooms, larger 'set piece' areas, like a maze or a crypt complex, will be pre-designed and will appear intact in the levels. This system facilitates the inclusion of puzzles and traps, and helps the addition of thematic elements. Deeper levels will contain progressively more difficult creatures and hazards. A character's quest must end with the defeat of Diablo, located deep in the dungeon"[1].

"Through the control of an objective function which can develop levels which meet with narrative or technical requirements."

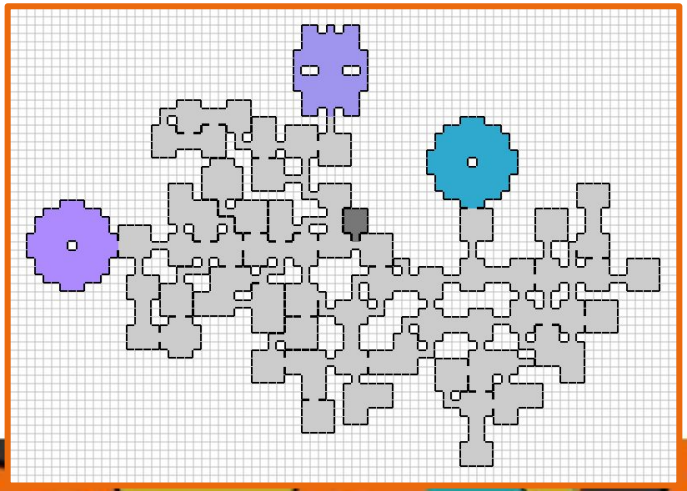
In reflecting on the success of *Diablo*, the DRLG would later be praised by those working on subsequent editions as being a major selling feature, due to such elements as the extended ability for replaying the game. Jay Wilson, lead designer for *Diablo III*, would remark that "games that have randomly-generated environments with randomly-generated encounters. Not easy things to do, but those things are key. It's what keeps Diablo interesting over time"[13].

To this end we developed a PCG placement algorithm for tiles. The construction is based off a Genetic Programming approach which controls the underlying connective graph. Through the control of an objective function which can develop levels which meet with narrative or technical requirements.

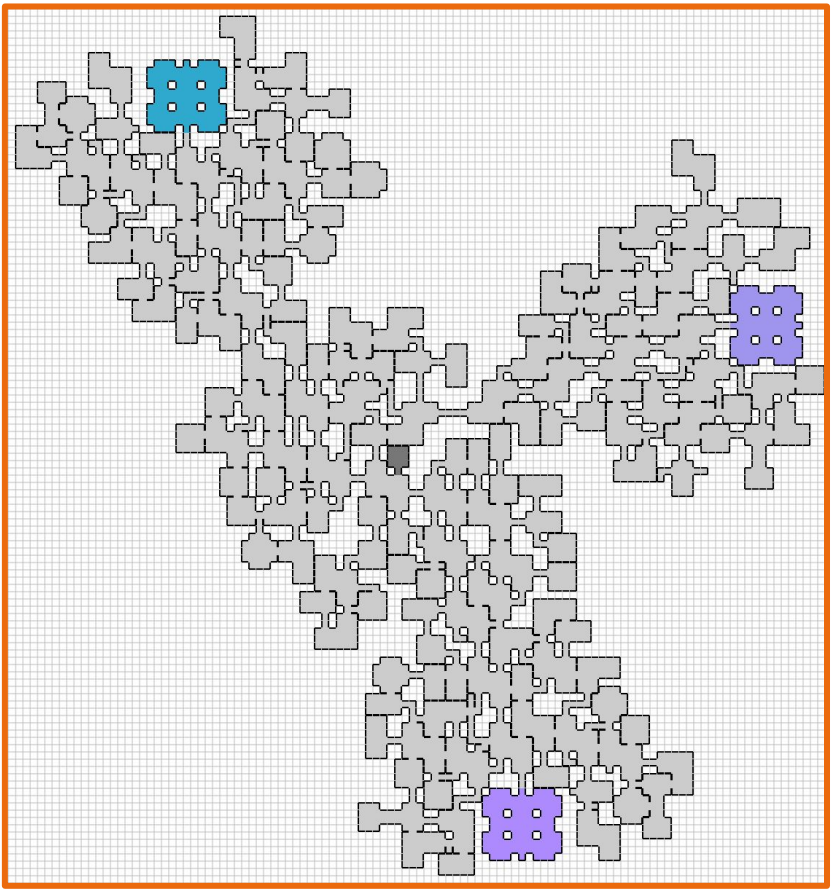
The first generation method aimed to have a compact room set with well-connected hallways with three boss battle rooms:



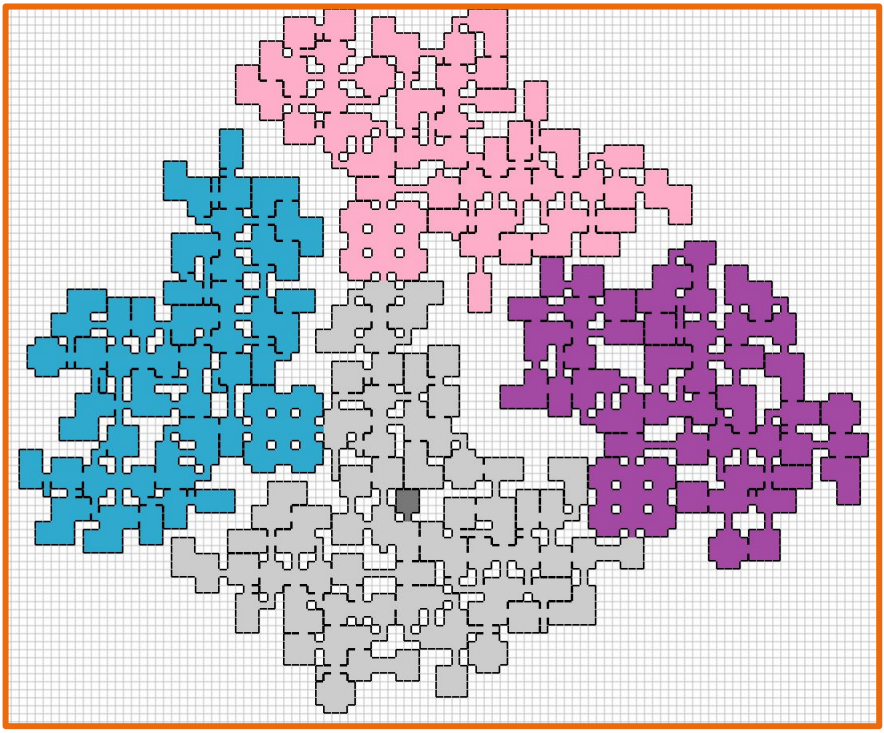
In order to show that on-demand generation could be achieved, we limited the size of the level to 100 rooms, created in no more than 500 evaluations of the fitness function, with the same requirements:



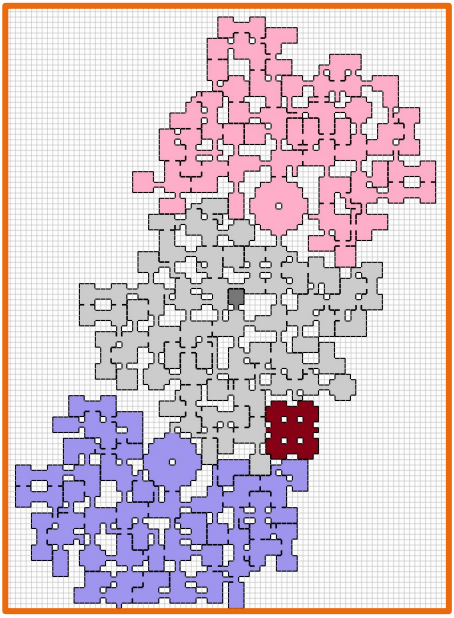
Third, we examined if there was a way to not only have boss battle rooms at the edges of the map, but also to maximize the distance between the each boss room.



Finally, looking at a number of games, boss battles act as gateways to other areas, so the last fitness function maximized the distance between the boss battle rooms, and then after this was generated, a second round of generation locked off sections of the map until after each boss battle:



This was not always successful as sometimes the boss rooms would be too near the start of a section, not allowing any further generation:



You can learn more of the technical details of the generator in [3] and in future work we would like to place some of these levels into a real game environment.

References

[1] Condor. Diablo: A game concept by Condor, Inc. 1994. Available at http://www.graybeardgames.com/download/diablo_pitch.pdf

[2] M. Lopez. “Blizzard's Jay Wilson Talks Diablo III” Game spy, page 2, August, 20, 2008. http://pc.gamespy.com/pc/diablo-iii/901260p2.html

[3] V.Valtchanov and J.A.Brown. “Evolving Dungeon Crawler Levels with Relative Placement”. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, pp. 27-35, 2012. https://tinyurl.com/seeds2Diablo

It's Out of Our Hands Now: Outsourcing Everything

By James Earl Cox III
@just404it

There is an adage that claims any creative project can be done fast, done well, or done cheaply, but you can only have 2 of the three. When making a small freeware game, or a prototype, having it done well may not matter.

"What if I made a project entirely through Fiverr?"

For a few projects, I outsourced music to people on Fiverr, a website where you can pay 5$ to people of the internet to do specific tasks. The music was never great, sometimes it was awful, but it was generally unique. As someone who can't make music (yet), having the ownership of a mediocre song that I can reuse in different prototypes is great. Then I had the idea: what if I made a project entirely through Fiverr?

I can make games no problem, so I wanted to outsource for something I had no knowledge of: a music video. One with a dancer, some nice background, lyrics, a beat. Your run-of-the-mill YouTube music video. But every step would be crafted and imagined by other people. I reached out to Fiverr for the following services:

- Background music
- Lyrics and singing
- Music video nature background
- Dancing

Total cost: $95. Not a lot considering all the different components and time spent.

To ensure I had minimal input, I waited for the previous step to be completed before I would enter the next. As such, I wouldn't ask for lyrics and singing until I could send them the background music. And I didn't ask for the Dancing until I had the track with both music and singing.

The only component I controlled was the core subject of the song. The lyric and singing artist required a subject for them to sing about.

Giving up such creative control led to a digital artifact that isn't quite anyone's vision, yet still exists. An Exquisite Corpse of Internet outsourcing.

The full music video can be experienced here:
https://youtu.be/YJw2RbEyqbU

It's wild.

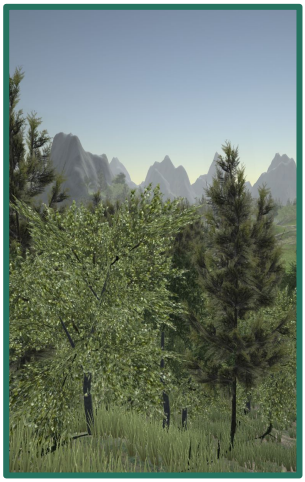
Making a Place to Wander

By Luke O'Connor
@lumoconnor | https://lukeoc.itch.io/

My entry into last year's ProcJam was A Place to Wander. It was intended to be a "pure" walking simulator, in which the player would explore randomly generated landscapes. In the absence of any narrative or goals, it was important that the landscape itself was interesting enough to engage the player. I wanted to recreate the feeling of being outdoors in a wide open space, where you are free to wander and move towards anything that catches your attention. The rewards would be the shifting soundscape and visuals, and learning the lay of the land.

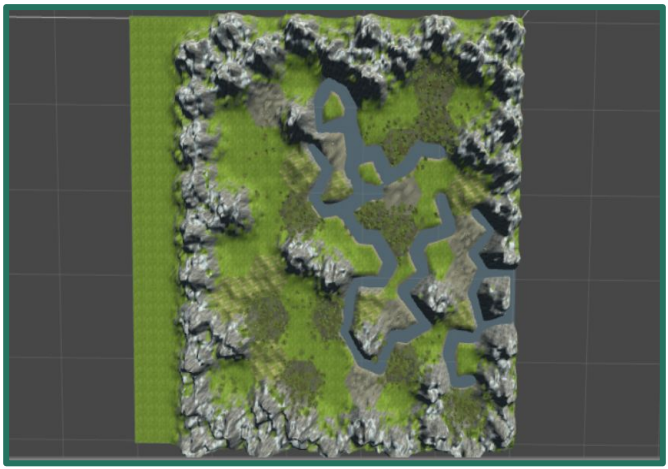
The main inspiration, in terms of interesting game landscapes, was Skyrim. The world of Skyrim is a diverse mix of mountains, valleys, wastelands, swamps, forests, coastline and rivers. Regions are distinct, yet blend together smoothly, and walking for 30 minutes feels like embarking on a grand adventure. How, then, would I use procedural generation to create such worlds?

The answer came to me from the world of boardgames. I am a big fan of boardgames, and growing up I particularly enjoyed any game in which you could build maps out of modular pieces. More recently, I had been playing Settlers of Catan, which ended up being the strongest influence on A Place to Wander's landscape generation.



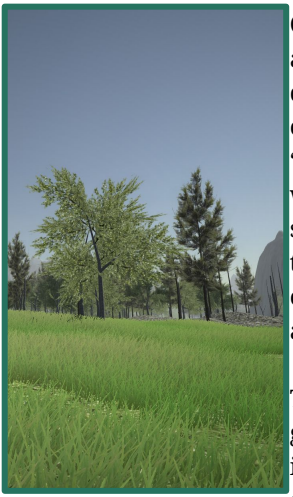
What I like about the maps in Catan is that by randomly placing the hexagonal tiles you would still end up with an island of diverse and distinct biomes, each of a similar size. I imagined building my worlds in a similar way, but instead of using predefined tiles, I would use tiles which are themselves procedurally generated.

The idea was that each tile could be generated using a completely different algorithm, making it very simple to produce a varied landscape while neatly bypassing some of the trickier aspects of tuning a more sophisticated algorithm. Mountains would be generated one way, rolling hills another. This approach is very extendable: tiles can be added or removed at will, each using a specialised algorithm to generate their terrain or geometry. It is also easy to mix the generated tiles with hand-made tiles.



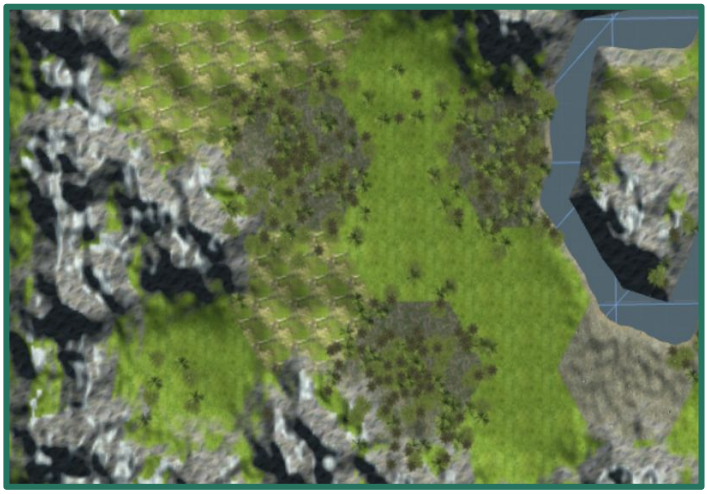
The underlying, tile-based approach was augmented with some blending between tiles, and a river generation algorithm that snaked its way along the tile boundaries. These “global” touches were simple to add and really tied the world together. That said, I think there is room for improvement in this area, and it is certainly a less trivial problem than the underlying tile based

algorithm.



Overall, I feel that this method was successful in helping me to achieve my goals for A Place to Wander. While the layout of tilesis quite clearly visible from a bird's eye view of the map, it is less obvious for a player wandering the landscape. Taking the tile based "boardgame" approach also paid off when building the soundscape, which was an important part of the experience. To produce a soundscape that shifted naturally as the player wandered through the world, I simply placed a different 3D audio source at the centre of each tile. I did the same with particle effects, to add to the atmosphere in each area.

This modular and boardgame-esque approach to procedural level generation has been successfully used by a number of games, including some of the most popular 2D Roguelikes, such as Spelunky and The Binding of Isaac. I believe that it is an excellent way to build interesting game worlds, and I would thoroughly recommend exploring its applications in 3D level design.



Procedurally Animating Horrific Abominations Without Wanting To Die

By Brandon Yu
@Chaoclypse

Limitations

I love body horror. I love Cronenberg, I love Shintaro Kago and I even have a folder on my computer that is chock full of body horror inspiration. I've wanted to make body horror feature prominently as a game mechanic for a while. However, as a student in DigiPen Singapore, I am often forced to be realistic. Students here have a few months to make a game each semester, often entirely from scratch in C++. Artists are usually overworked as-is, and my game development group in particular has no dedicated artist. Doing an animation-heavy game would be foolish. However, thankfully, limitations can often be a blessing in disguise.

"I needed something that was modular, but also believable."

I am currently working on TRASHBOMINATION, a game about shooting limbs off procedurally animated abominations. I needed something that was modular, but also believable.

The reason that it had to be modular was to facilitate dynamic removal of limbs - the reason it had to be believable was to make it as freaky as possible. The procedural animation system made it really easy to make new abominations, while still being able to surprise me.

"This may not be the most elegant system, but it is trivial to implement and easily supports dynamical node removal."

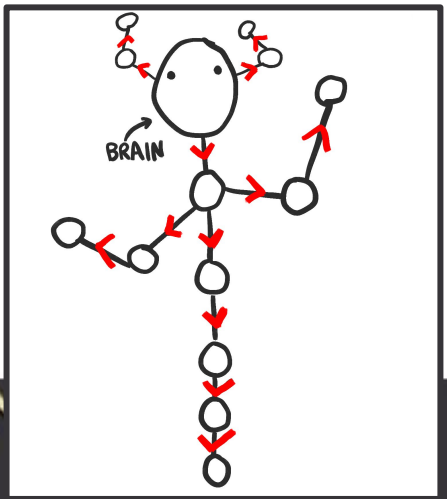
The abominations themselves were not procedurally generated, but understanding how they were built to be procedurally animated makes procedural generation trivial.

This being a zine, it's hard to show them in motion, but you can see gifs of the abominations moving at my Twitter. (https://twitter.com/chaoclypse)

It's Alive!

The way that the abominations work and move is actually very simple, without even the usage of inverse kinematics. They use nodes, combined with a messaging system. Each node is only aware of its parent and its children. This may not be the most elegant system, but it is trivial to implement and easily supports dynamical node removal.

Here is a diagram of the layout of the abomination – the arrows point from the parent to child.



The “brain” is first initialized, which then allows it to control every other node via the messaging system. The message can flow “up” (sent to the parent, who sends it to its parents, and so on) or “down” (sent to all of the node’s children, which send it to their children, and so on).

Whenever a message gets passed to a node, it knows:

1. What kind of message it is (Movement, Translation, Rotation, etc.)
2. Who sent the message (Parent, children?)
3. What the additional parameters to the message are (A struct with additional information)

Given this information, it then calls a function that can interpret that particular message and passes along the relevant information. This function is a virtual function, which means that the nodes can use inheritance.

A Variety of Parts

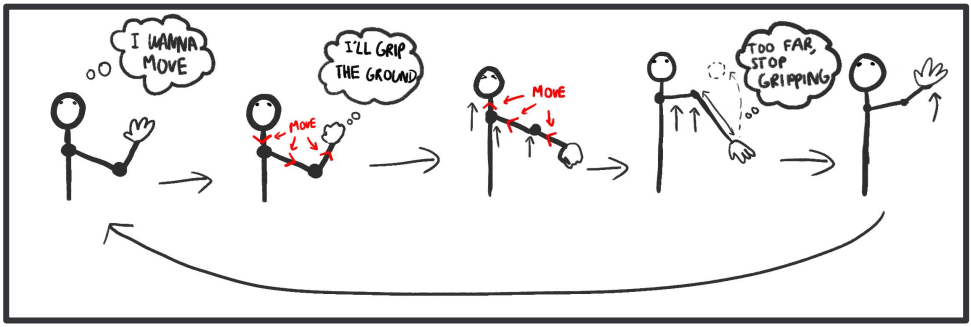
The nodes currently in TRASHBOMINATION are the brain, the spine, the elbow, the hand and a “tail” that allows for some rope physics-style action. They all act differently given different messages and they all also have their own update loops. They also utilize inheritance – for example, the elbow can inherit from the spine since it acts similarly apart from maintaining an orthogonal angle from the parent. The “base” part that they all inherit from has a variety of convenience functions (such as translation or checking distance to parent part).

“They all act differently given different messages and they all also have their own update loops.”

One important thing all of them have is a function that is called when the distance from their parent exceeds a certain value – in the case of the spine, it just closes the distance, in the case of the hand, it moves orthogonally to the direction from spine to elbow.

Moving Along

As an example, the way the abomination moves is similar to how real animals move. They grip on the floor with their feet, propelling the rest of their body forwards.



The brain first passes down a MOVE message down the spine, onto the elbow then finally onto the hand.

The hand then, if possible, grips on the floor, and propels the body by sending the MOVE message upwards through the body.

The parent nodes will then note that the MOVE message was sent from the child and then move in the specified direction (Other nodes that are too far away will be notified, and will act accordingly). While gripping, the hand remains stationary.

This goes on until the elbow moves too far from the hand, at which point the hand then uses simple vector math (I used a reflection transformation) to figure out the next "grip spot", which it then moves towards. The hand does not cause the body to move when it is not gripping the floor, because that wouldn't make sense.

And that's it! Super simple, but effective.

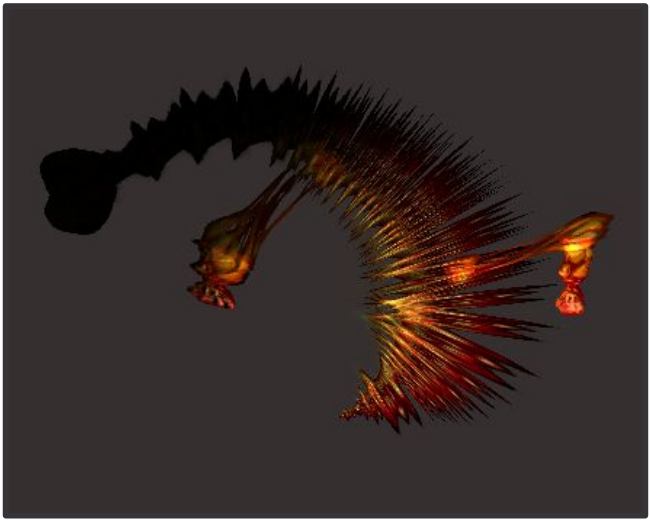
"But if the rhythm is even slightly out of sync, it would be wise to include a feedback which nudges our runner back into the unstable standing state, before it is too late."

Abomination Factory

A major advantage to using a modular, node-based system is that a large variety of abominations can easily be created. You can even create the abominations procedurally if you want – any node is compatible with any other node. And of course, they are dynamically destructible, as well. This allows you to do a bunch of cool tricks. A few ideas:

1. Bunch the nodes close together in order to create an extremely fluid, almost ropey body.
2. Vary sizes over time to create “tails” and so forth.
3. Create a completely “stiff” elbow (keeps the same angle with the parent at all times) and use it to create decorations on your abomination.
4. Use the ropey bodies in (1) and combine them with the stiff elbows in (3).

Here are a few examples of some of the abominations you can create with said tricks – all are working, walking abominations made in under 30 seconds each. They can look pretty snazzy, even when completely static!



Conclusion

Procedural animation is great, and extremely versatile. However, the way you implement it should stem from the feel you want the creatures to have in-game. If you want to do a more precise kind of movement, then inverse kinematics is great, but it does come with its own problems. This is just one simple way to go about it – be sure to explore!

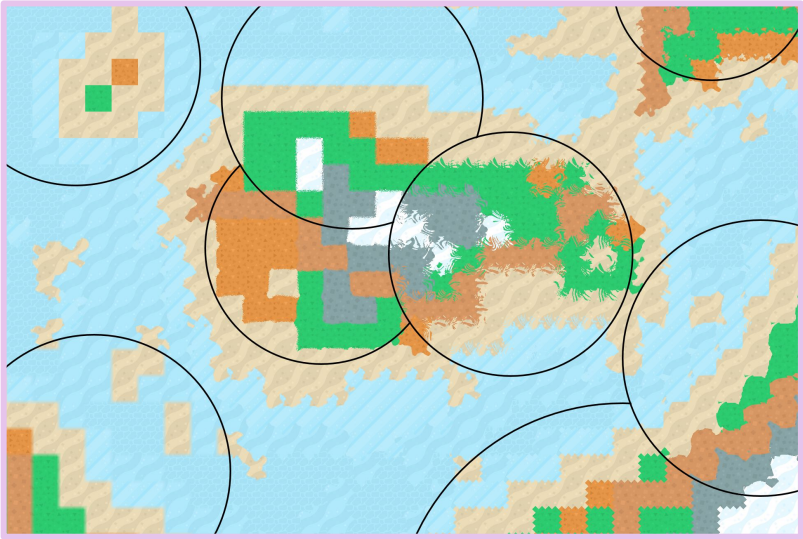
Wangscape

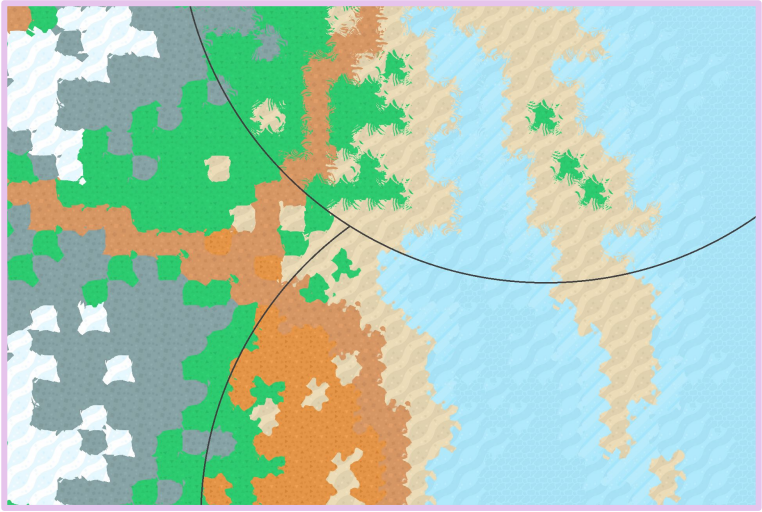
By Serin Delaunay & Łukasz Hryniuk
@SerinDelaunay | https://github.com/serin-delaunay/ | https://github.com/hryniuk

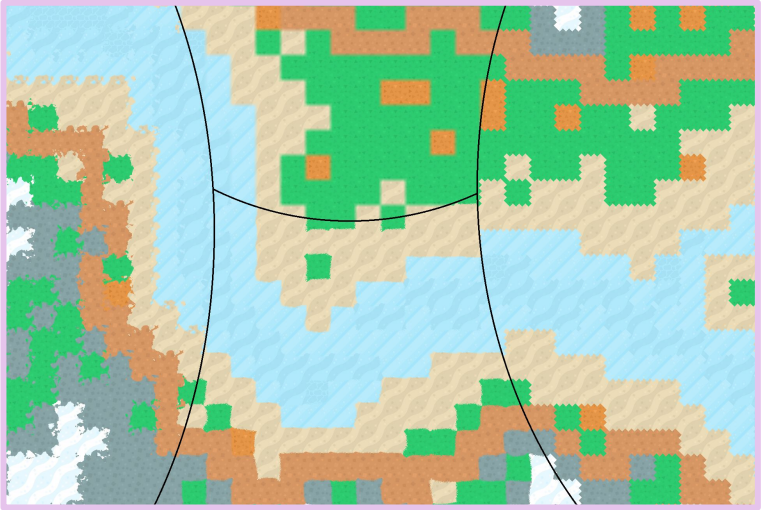
Wangscape converts rectangular terrain tiles into corner Wang tilesets with procedurally generated borders. These examples use CC0-licensed tiles by @KenneyNL.

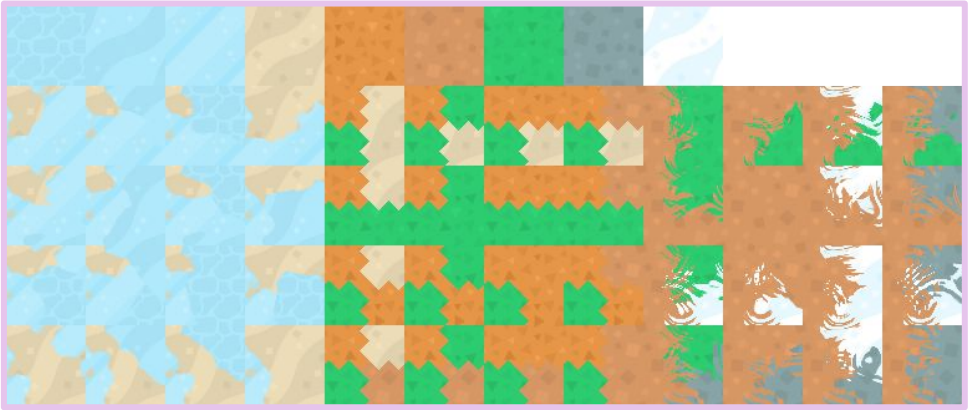
Wangscape can be configured to make borders which are plain, subtle, weird, or even extravagant! It uses libnoise modules (http://libnoise.sourceforge.net/, http://www.restnoise.org) as building blocks to blend textures together, which can be configured by JSON files.

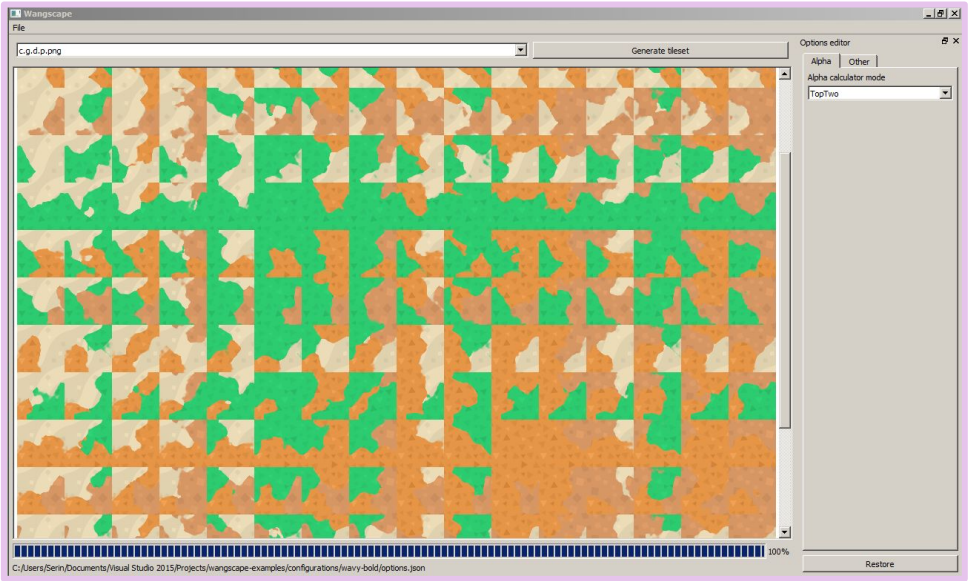
You can try this with your own tiles, too! Download Wangscape for free at https://github.com/Wangscape/Wangscape. It's under active development by Serin Delaunay and Łukasz Hryniuk, and we're looking for new contributors. Currently we're working on a graphical user interface for easier configuration, and support for isometric tiles.










Wangscape
File
c.g.d.p.png
Generate tileset
Options editor
Alpha
Other
Alpha calculator mode
TopTwo
100%
Restore
C:/Users/Serin/Documents/Visual Studio 2015/Projects/wangscape-examples/configurations/wavy-bold/options.json

Goblin Frontier

By Matthew Santacroce
@AdventureByte | http://goblinfrontier.com/

Exploring the Frontier

Goblin Frontier is an open-ended adventure RPG game which focuses on exploration, base building, and survival in the harsh wilderness of a mysterious land. The game seeks to bring together the excitement of epic adventure games with the satisfaction and reward that can be felt when you create a homestead out of a once overgrown and wild piece of land.
The game will allow for houses and farms to be built by the player, these can eventually be called home for both animals and NPC's alike.

Crops will need to be planted and harvested if one expects to survive on the Frontier.

The Frontier is a dangerous place and it would be unwise to travel alone, particularly at night. Goblin Frontier offers split screen cooperative play, so bring a friend or tame a wild beast to serve as your companion and adventure together!

The game features procedurally generated terrain so that you will always have a chance to experience something new and discover interesting locations. Travelers roam the countryside. Tribes of old and creatures of myth have been spotted at the far reaches of forests and dark depths of caves. Wizards have been said to build towers that pierce the sky. Merchants hide treasure amongst the hills. The Frontier is full of unique, handcrafted places and you will never know what lies

Concept Art by @SecondSanta

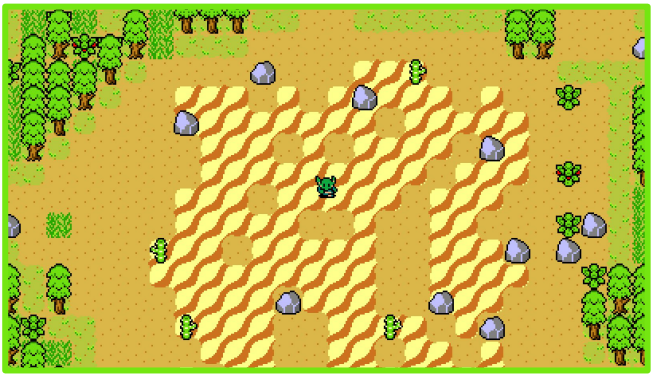
around the corner!

Putting the Pieces Together

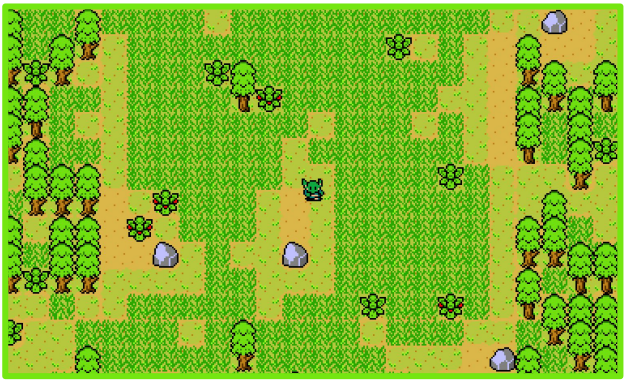
Two notable techniques I have utilized thus far in development are Perlin noise and Cellular Automata, both of which have many variations that can be implemented in a game world.

Perlin noise has been particularly useful in the generation of terrain in an expansive world. By overlapping two layers of Perlin noise, say one that returns a height value and another which returns a temperature, it becomes possible to create a variation in adjacent biomes. These biomes are specified based upon any preset combination of values (height, temperature, moisture, etc.) returned from such functions. An example would be passing an X and Y coordinate pair into separate noise functions, values ranging from 0 to 100 are returned, which in turn define a given grid block's biome or region designation.

Typical outcomes might be a "Forest" from a mild temperature and moderate elevation or a "Mountain" from a lower temperature and larger elevation.



Cellular Automata have been used for all kinds of things ranging from creating bodies of water and interesting cave networks to groves of trees surrounding open fields. By indicating an initial spawn chance tied to biome type, a grid of zeroes becomes partially filled with ones. Then a specified number of simulations steps are carried out on that grid which either add or remove grid values based upon neighboring cells. The grid is used to generate objects in the game world

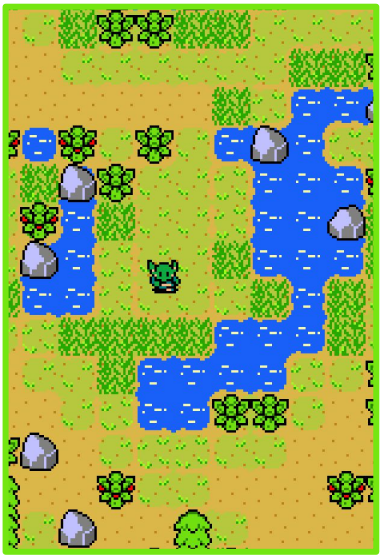


based upon those final grid values. The starting simulation parameters (initial spawn chance, growth and decay rates of the simulation) are best found by experimentation. I recommend starting with more extreme initial values and gradually changing to those in between until an adequate starting value is found.

"I recommend starting with more extreme initial values and gradually changing to those in between until an adequate starting value is found."

The beauty of procedural generation is that it allows for a sense of surprise to be felt by all of those who play the game. It offers up unique experiences which when coupled with an open and dynamic world help shape unscripted narratives that players can enjoy and share with others. The excitement of the unexpected outcomes is even felt by those who create procedural content. This game was made possible by the awesome power of procedural generation and it is my hope that it can serve as a memorable adventure for those who get a chance to play it alone and with friends.

Sharpen your spear and ready your shield, the Goblin Frontier awaits!



Embrace your Generator's Limitations

By Tom Coxon
@tccoxon

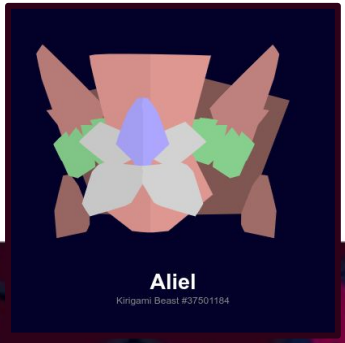
Content generators can often be complex. This is frequent especially when there are stringent requirements for the content: it must meet a specific art style, or satisfy a particular connectivity constraint, for instance.

If you have the luxury, it can help to reduce complexity by reversing the requirements between your generated content and the surrounding context. Decide what type of generator you need first, figure out what sort of limitations and problems it is likely to have, and *then* make other design decisions.

"The idea for this generator came when I realised a very simple recursive algorithm could produce a wide variety of body shapes."

This was the approach I took when planning Kirigami Beasts, an online procedural art tool that creates Pokémon-like monsters (http://kirigami-beasts.com/). The idea for this generator came when I realised a very simple recursive algorithm could produce a wide variety of body shapes. I knew from the outset that small details would be a challenge with this approach, and that it would make pixel art totally impractical and ugly.

So I designed Kirigami Beasts around its generator's limitations, and chose a minimalist vector art style. With this style, it doesn't matter too much if limbs don't fully connect to the body, and I can completely sidestep the problem of generating suitable textures. The viewer's mind can fill in the details much better than any algorithm could.



While it is easy to tweak presentation to suit the generator in an isolated procedural art tool, selecting contextual details to complement a generator's limitations does happen in full games too.

Mossmouth's Spelunky generates platformer levels in a grid of rooms. Although it generally tries to ensure connectivity between rooms, there are some room templates where walls can end up blocking off the path. This happens with a low, but non-zero probability every time a new level is generated.

Image courtesy of Darius Kazemi's article on Spelunky's level generator (http://tinysubversions.com/spelunkyGen2/).

The dim red tiles in the middle are probabilistic tiles that can randomly be either wall or background. This template is *supposed* to connect the left and right sides of the room, but there is a small chance that a vertical slice of these tiles will all become walls, preventing movement through it.

Rather than trying to provide connectivity guarantees by either constraining templates or complicating the generator, Spelunky allows the player to use bombs to break through walls. Bombs

simultaneously compensate for the generator's inability to guarantee connected rooms, while adding variety to the gameplay.

Other times, when you're lucky, the context for your generator is coincidentally already set up to complement your generator's limitations... That was the case with Starbound's procedural quest generator.

In Starbound, villager NPCs offer players procedural, multi-stage quests. The individual stages relate to each other causally, and each stage can involve a variety of different tasks. It is more sophisticated than Skyrim's Radiant procedural quest system in a technical sense, but it probably would not have been suitable for use in a game like Skyrim.

"The system cleverly knows how items work together and fit into quests, but the end result is... ridiculous."

An example of a whole quest an NPC might give the player is:

1. Take these seeds and grow me 2 tomatoes, 1 sugar, 1 corn.
2. Use 1 tomato, 1 sugar, 1 corn to make some relish.
3. A monster snatched my bread! Can you hunt it down and bring it back?
4. I loaned my raw steak to a friend. Ask him for it back!
5. Finally, use all that to make me a hamburger, I'm famished.

The system cleverly knows how items work together and fit into quests, but the end result is... ridiculous. This quest sends the player on a 5-part epic to build burger relish from scratch, and combine it with meat and bread that it (unintentionally) hints have already been digested!

We did end up introducing some extra rules to the generator to prevent it doing some *very* stupid things. However, trying to prevent all cases of stupidity would have either severely limited

its variety, or added a huge amount of complexity to something that was already overly complex.

In a game like Skyrim, which has a serious tone, this quest generator would have been canned. In Starbound however, NPCs already have a whimsical and silly nature. Villagers occasionally set themselves on fire with hand dryers; they vomit after using a teleporter to move six feet; and they fall in love with other NPCs and follow them around with goofy grins. It makes perfect sense for a silly NPC to give a silly quest.

In the case of Starbound's procedural quest generator, the limitation was its complete lack of common sense. Thankfully the generator's context—silly NPCs—perfectly complemented the generator's stupidity.

If you have the luxury of control over the context of your generated content, keep your generator simple, and turn its shortcomings to your advantage. With the right approach, even a design flaw can become a cool new feature!

A Conversation Between Botmakers: Part 1

By Heather Kelley & Audrey Moon
@PerfectPlum | @animalphase | http://www.perfectplum.com/ | http://loveme.computer/

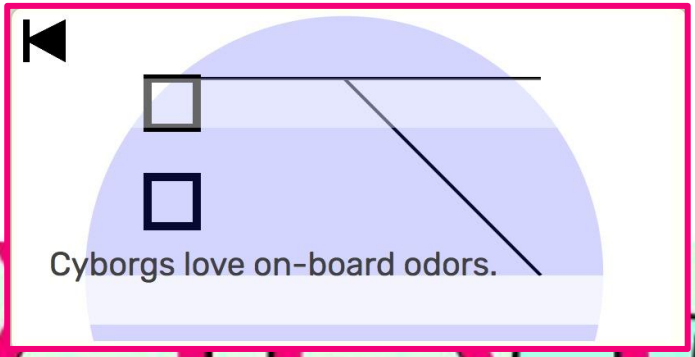
Editor's Note: *We've tried to keep the raw, IRC vibe of this conversation between two botmakers intact, and have only lightly edited it in a few places, but we've also trimmed it for size. For the full unedited interview, featuring lots of reflections on AI, futurology and science fiction, check out http://www.procjam.com/seeds/*

@indulgine, the Indulgent Engine, is a twitterbot by Heather Kelley and Audrey Moon built with cbdq and tracery. @indulgine wants to remind us all the different ways machines sense and feel pleasure. Its slogans are embedded in SVGs inspired by mid-1960s computer-generated art, particularly the work featured in the 1968 ICA exhibition Cybernetic Serendipity. We interviewed each other about the creation of the bot!

[@perfectplum (PP)]: What appealed to you about the original concept for @indulgine?

[@animalphase (AP)]: I was really interested in playing with javascript and generated SVGs. We also found so much synergy on the theme! From my past works and interests, even to my personal domain (loveme.computer) we both had a thing for thinking about pleasure, non-organic machines, AIs and other consciousnesses.

PP: Yes, and I didn't even KNOW that was your domain when I asked you to collab, so it must have been some kind of weird mind meld thing!



AP: I was also really excited to work with you! You seem really cool, do a lot of cool work, and collaborating was a blast even though the timing was weird. I'm curious what started this whole project/talk/idea, since you concepted it before bringing me on?

PP: I was invited to speak on a panel at a Google Design conference [1], along with some artist colleagues of mine from Carnegie Mellon. And they wanted us to talk about humans' changing relationship to machines/AIs/whatevs. But also a bit like an artist's talk. My work in games is often about the senses, so I started thinking about how that relates. And then decided that instead of doing a standard talk I wanted to make a new thing. I've been meaning to make a bot for about a year but there are so many cliches in twitter bots and i held off because i didn't have a good idea.

AP: Do you have any strong feels about senses of machines?

PP: Well the strong feeling i had is that if we expect they will be sentient/intelligent, it seems really unethical and downright stupid to not think about their sensory lives. Like why replicate the false mind/body dualism? Also i have a categorical inability to take anything completely seriously so i wanted to be a bit tongue in cheek. I'm also interested in expanding human senses so thinking about what senses machines will have that are different from our "natural" senses is one avenue to that.

AP: Yes, I'm totally down for that tone! I tend to have a lot of serious conversations and make a lot of important statements with a tongue-in-cheek tone. I just… think it's fun to play with concepts and enjoy them even while taking them very seriously and treating them with sensitivity.

PP: Yes that! Like, humor doesn't mean, not serious?

AP: Yeah, totalllly. Obviously, it can be a challenge to handle things with that tone and not everyone's down with it! So it's cool we vibed that way.

PP: Humor hopefully disarms and gets past people's defenses. Like, i can say something silly and laugh about it but still 100% mean it. I mean, I'm a game designer, after all. I play. Play is where we explore and discover.

"...Data made a child and gave that child time to try on all kinds of different appearances of species and genders to pick the one for them and it was so wholesome."

AP: We are all homo ludens after all lmao. Maybe we'll build machina ludens.

PP: I mean GOD it would be awful to be an AI that was unable to enjoy anything. And not permitted to feel anything. It basically reminded me of our colonial perspective on any "other".

AP: Yes totally! I've also been rewatching TNG with my nbfriend and Data's character arc has been a key focus for us! Like, finding their own ways to enjoy things from their own perspective, tho still trying to understand why humans enjoy things they way they do too.

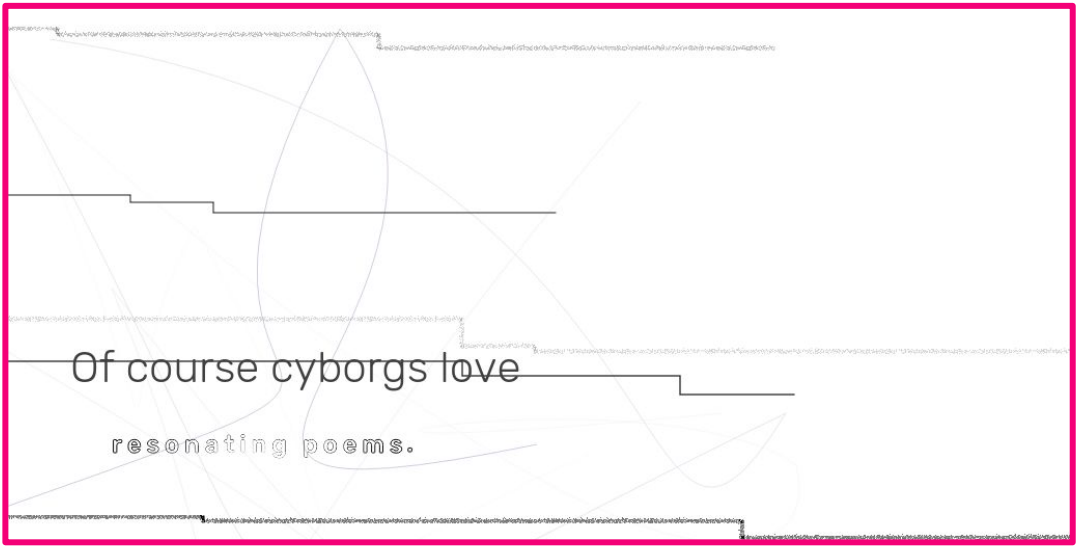
AP: Data is good and pure and a great trans/nonbinary icon for us lol. Like, so many similar struggles, having to have pronoun conversations with the captain, making sure people use their real name instead of pronouncing it incorrectly. And also like, Data made a child and gave that child time to try on all kinds of different appearances of species and genders to pick the one for them and it was so wholesome.

PP: It's hard to imagine how forward-thinking that was at the time.

AP: But yeah, it's kinda interesting thinking about how that character arc could be redone today, especially looking forward to what we're talking about here in the project! There's like, example of the enterprise crew encountering new lifeforms or accidentally

creating species of nanobots that achieve sentience and going through many efforts to find a way to communicate to establish a happy and mutually-beneficial relationship. Like, if an ai enjoys being alive, how would it think about itself?

Find part 2 of the interview further on in this issue on page 130!



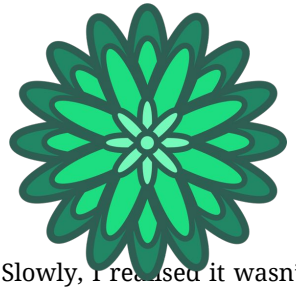
References
[1] https://design.google/span2017/pittsburgh/

Emergent Tools

By Mark Rickerby
@maetl

"It opened up the possibility of shaping a world around the characters using the epistolary fragments."

Once upon a time, I tried to write a traditional novel. I spent years amassing a manuscript of character perspectives, world building notes and epistolary fragments that stretched to more than 120K words. I had a strong vision for the story and the motivation to tell it, but it never felt right as a whole.



Slowly, I realised it wasn't the characters or narrative I was struggling with, it was a more fundamental problem of allowing the story to shape its container. The structure I wanted made more sense as interactive fiction. Forcing it into the constraints of a traditional page-bound experience was the root cause of my frustration.

This realisation was immensely freeing. It opened up the possibility of shaping a world around the characters using the epistolary fragments. I was originally intending to discard. Through emails, documents and fake web pages, I could allow readers to inhabit the desktops and phone screens of each character and see the world through their eyes.

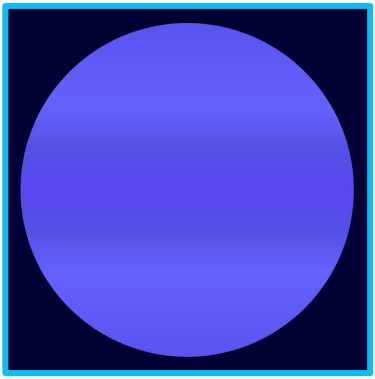
Things became frustrating again when I realised I'd need a lot more content to make the fake desktop experience feel continuous and cohesive. This led me to wonder… What if I didn't have to manually write it all? Could I generate expressive text automatically?

By this long, strange, arduous route, I discovered procedural generation and generative methods.

§

I had no idea what I was doing

at first. I vaguely knew that generative grammars were useful for creating sentences with many variations. I plugged several Ruby gems into my app to try this out and it worked!



I was really happy with the initial results but soon started running into problems. The gems were unmaintained and rough around the edges with poor data structure choices that led to errors and limitations. Attempting to fix them introduced even more problems.

So, starting from scratch, I wrote a tiny parser and abstract syntax tree to support basic template expansion and ported all the existing generators across. Now I had my own grammar tool. Something I'd never planned to create but it was doing exactly what I wanted.

As I used it more and more, I fixed bugs, ironed out edge-cases and added tests. Eventually, I accepted this project-inside-a-project was moving far beyond mere yak shaving. I refactored the internals, named it Calyx (http://calyx-rb.org), and published it on Rubygems.

```
planet.generate(
  seed: seed.to_s,
  base_fill: '#DDCA7D',
  overlay_fill: '#DDCA7D',
  blend_mode: 'normal',
  base_frequency: '0.005',
  num_octaves: '6',
  noise_type: 'fractalNoise',
  alpha_shift: '0.1',
  terrain_opacity: '45%',
  diffuse_lighting_color: '#C3423F',
  diffuse_constant: '3',
  diffuse_surface_scale: '20',
  specular_lighting_color: '#C3423F',
  specular_constant: '1',
  specular_surface_scale: '5',
  surface_lighting: 'combined'
)
```

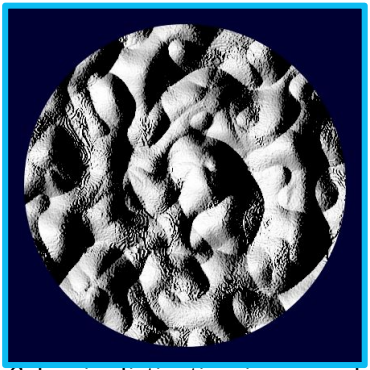
I entered #NaNoGenMo and used Calyx to make a weird generated fantasy novel. Writing grammars for a much larger body of text surfaced usability problems which led to further improvements like weighted probabilities, memoization, and repeatable random seeds. I even started getting feedback on GitHub. Other people were using it!

At this point, it might seem like things were going pretty well, but I wasn't sure at all. I started overthinking. I learned about the history of generative writing and witnessed the extraordinary success of Tracery and its inspirational creative community. Grammars were far more widely used and studied than I'd known. If I had understood this at the beginning, maybe I wouldn't have needed to make Calyx at all.

"Perhaps this existential questioning is inevitable when something that started out as a means to an end becomes an end in itself."

Was I wasting time duplicating things that already existed? Was it procrastination, avoiding the difficult effort of finishing my Novel?

Perhaps this existential questioning is inevitable when something that started out as a means to an end becomes an end in itself. In order to go from confusion and self-doubt to clarity and confidence, I had to think about the balance between what exists in other tools and what Calyx uniquely has to offer.



Calyx is distinctive in several ways: its Ruby DSL, support for weighted probability distributions and the expression syntax for memoizing and cycling through expansions. Interestingly, results can be returned as s-expressions of

expanded nodes as well as plain strings, so in future, there's no reason why it couldn't evolve beyond text into a general purpose tree structure generator.

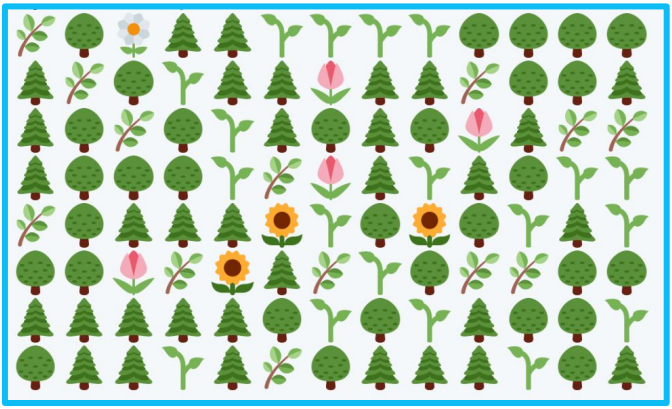
§

In programming, we have a tendency to get far too hung up on the problem of 'reinventing the wheel'. It's an attitude that largely comes from the imperatives of commercial software. Yes, it is undesirable to duplicate effort by copying an existing library or tool. It's also one of the best ways to learn.

This cliché also elides the distinction between the abstract concept and the concrete reality. There are as many types of wheel as there are vehicles. Wheels need different affordances for different conditions. Context matters.

In the process of struggling to produce an enormously complex and flawed creative work, I developed a tool with unexpected utility.

More importantly, I discovered a whole new community of people doing fascinating and fun work on the fringes of games, compsci, art and literature. Thanks to everyone for your encouragement, enthusiasm, and willingness to share.



Let there be a Graph!

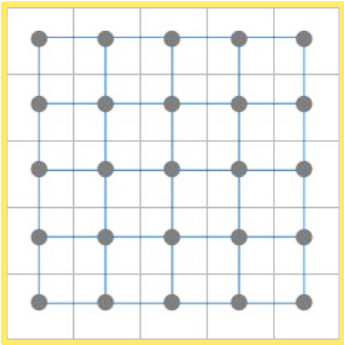
By Ahmed Abuzuraiq
@ahmedabuzreq

Say you have a level in your game that you want to divide to regions, only that you want to control which regions should adjacent or nonadjacent. That might because you decided that if the player want to go from a region A to another region C, she must first pass by B or maybe like in the game RISK the adjacency between the countries and continents influence the player's strategy and you want to put that in consideration when generating the maps on which the game will take place.

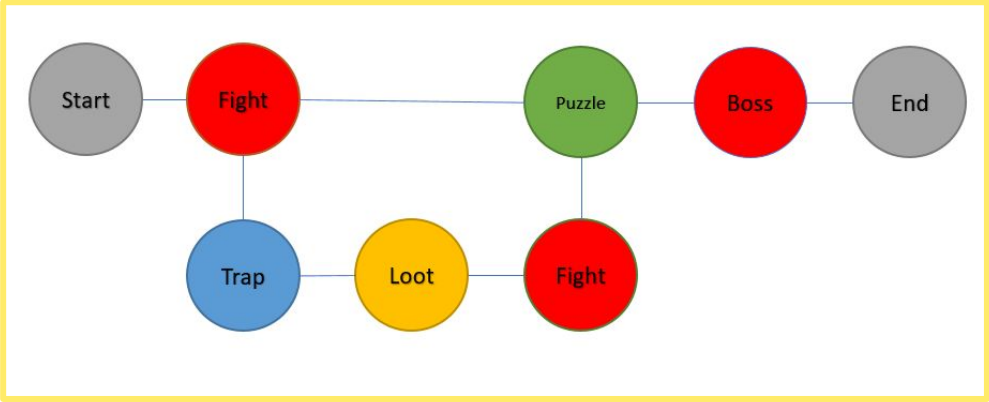
I can help you in doing that! But I am going to ask you for two things, first I want your level or map to have a graph representation. (you know nodes and edges) let's call this the Basic graph. So, if your level is a simple rectangular grid then the cells will be represented by nodes in the Basic graph and if any two cells are adjacent then we will create an edge between their nodes. You can also use something more fancy like the graph of a Voronoi diagram.

What I will do is that I will try to divide the Basic graph you give me so that the portions of the Basic graph resulting from this division are adjacent in the way that the Constraint graph describes.

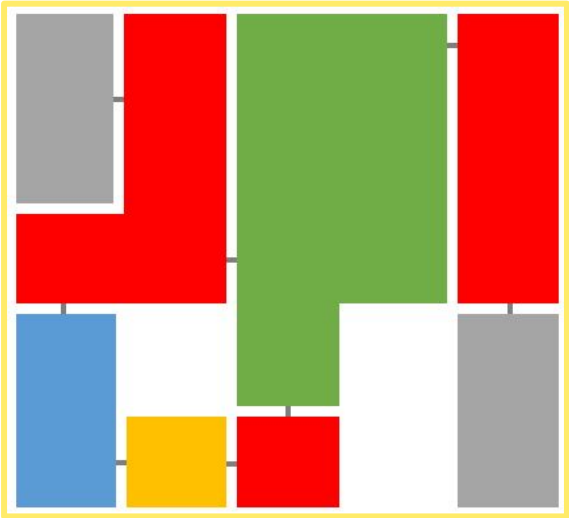
Let me be more concrete here:
This is a Basic graph, a simple 5x5 grid.



This is the Constraint graph.

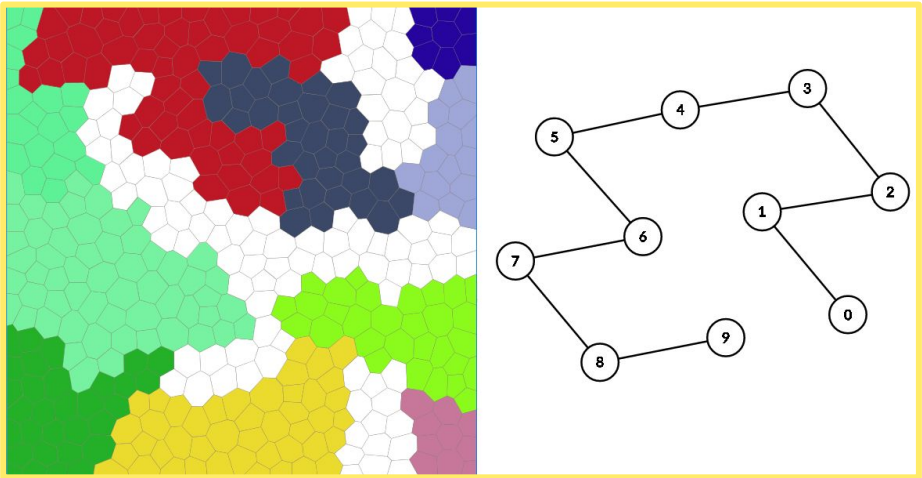


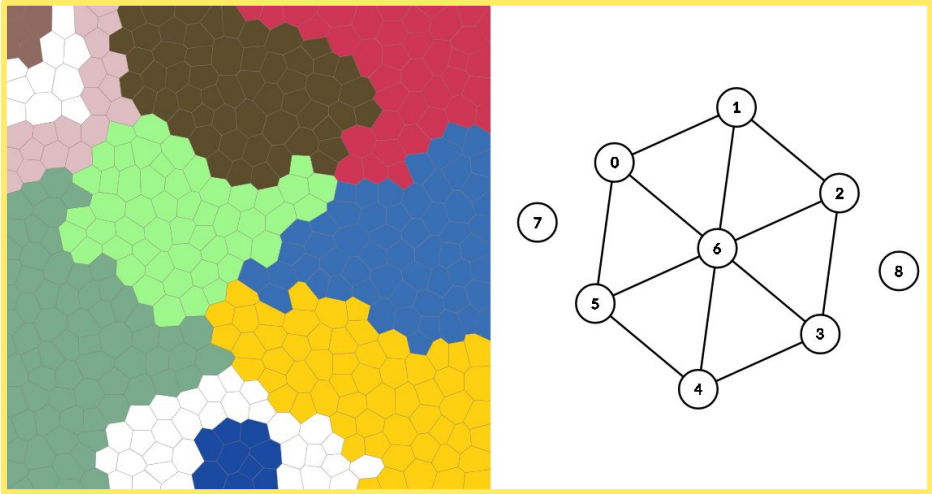
The result is a partitioning of the Basic graph where the regions can each be “mapped” to one node in the Constraint graph (this is called graph isomorphism, iso = equal, morph = shape)

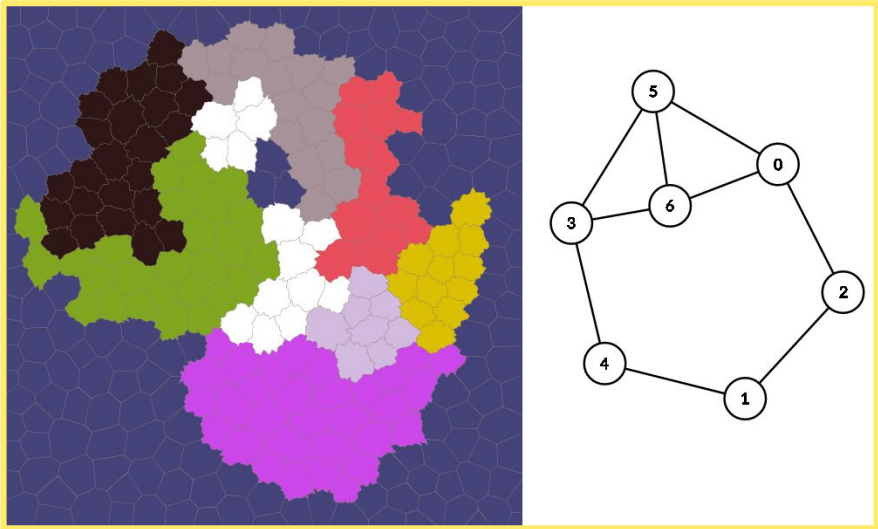


Compare this with the Constraint graph and you can see the mapping!
I bet you wondered: what's with those cells you colored white? well those are completely removed from the Basic graph, it just makes everything easier if I am given the freedom to do that.
Also, you might have read the labels on the Constraint graph, well in this case I am using the Constraint graph to divide the level to meaningful regions, for example I wanted to put the boss region before the ending region and a trap before the loot region.

Now I will show you more of this stuff, I will be using a Voronoi diagram that I generate randomly this time and the Basic graph will describe the adjacency between its cells. The result on the left and the Constraint graph on the right. The cells colored with white are removed from the Basic graph.
I will leave figuring out the mapping as a small mental exercise for you!







This last one is an island; the blue cells are water and are not part of the Basic graph

Finally, you can use this tool and know more about it here: https://github.com/abuzreq/ConstrainedGraphPartitioning

Rogue Process' Skyscraper Catalogue

By Mike Cook
@mtrc

Rogue Process is an action-hacking game about slow-motion hacking and high-speed skyscraper parkour. You play as an acrobatic freelance hacker who makes a living sneaking into corporate offices and stealing their darkest secrets. Naturally, we procedurally generate all those skyscrapers - this is a quick overview of how I go about doing it.

"Rogue Process' generator actually makes a *lot* of mistakes - sometimes hundreds per building."

When I was building Rogue Process' skyscrapers generator I wanted a generator that had a lot of freedom. Some generators are *constructive*, meaning they're methodical and smart enough to not make mistakes, but these generators can also be a little bit more predictable. Rogue Process' generator actually makes a *lot* of mistakes - sometimes hundreds per building. But it's fast enough to quickly restart and try again, and we have some code that makes sure we don't fail too many times before bailing out and using a safer option.

I also wanted a generator that was easy to extend and add bits to. Spelunky's level generator uses level chunks made by its designer, and glues them together like a jigsaw. This is great because adding a new chunk to the database is simple, and adds loads of value because it can be used in so many new combinations. I wanted something similar, but Spelunky's approach didn't fit well with man-made structures like skyscrapers. Instead, I went for something a bit looser.

With that preamble out of the way, here's a quick overview of how we generate buildings:

1. The Outline. We start with a rectangle of space we're going to put a building in, but we rarely end up with rectangular buildings - the generator either randomly bites chunks out of the edges, or it picks a special template to build within, like a building with a sloped roof.

2. The Chunks. Chunks are like zones in SimCity, or the ghost of where a room will go. They let us think about the flow and makeup of a building without actually committing much detail. The generator marks out rectangular shapes in the building (from a small 1x1 room up to huge 1x5 or 4x3 rooms) using a catalogue of shapes for this particular level. This means that Industrial Sector buildings can have huge warehouses and maintenance shafts, whereas little executive offices mostly have simple corridors and meeting rooms.

3. The Rooms. In this phase the generator tries to find rooms that fit into each chunk it placed in the previous step. Each room has a specification that describes where it can be used: there's a bit of code that checks if a placement is legal (for example, a CEO office must be on the top floor); a bit of code that checks if a link between rooms is legal (for example, it's not okay to build a staircase up into the bottom of an incineration shaft); and a bit of code that describes how to decorate the room. The decorators are usually several bits of code connected together (I use something called Delegates in C#) which means a decorator is like a buffet for each room. Each room has a collection of generic decorators (most want a floor, some will want windows) and then some special ones (like placing a freight elevator or worker drones).

"Each of these placement phases has a budget it can spend on adding stuff so we can make sure it doesn't put too much of one thing in a building."

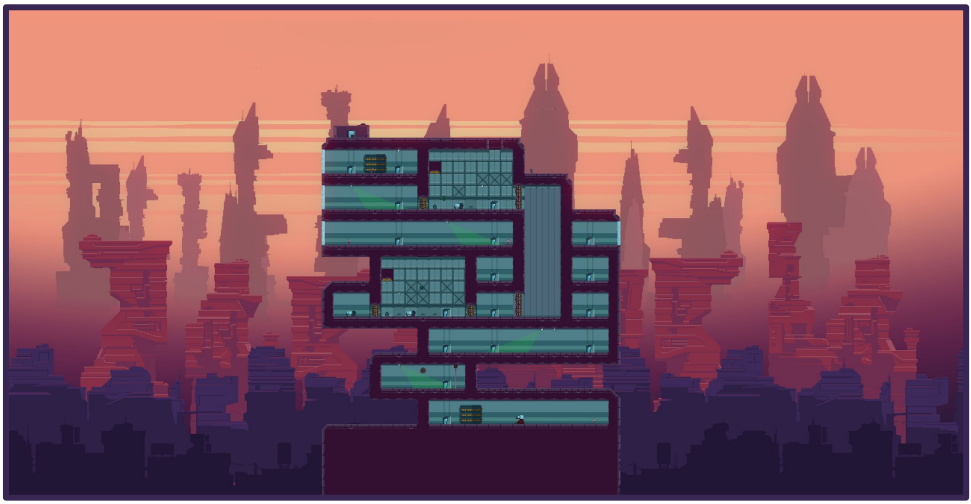
4. Sewing It Up. As the decorators work, they also place useful markers to indicate things about the building. For example, they build a map of the building that NPCs can use to walk around it, and they mark out places where secrets might be hidden. The final stage then goes over the finished building and adds all the stuff that isn't specific to any single room: key cards, datacenters, guards, security systems. Each of these placement phases has a budget it can spend on adding stuff so we can make sure it doesn't put too much of one thing in a building.

Each stage of this generator is really easy to interrupt and get

hands-on with: we can supply partly-finished outlines if we want our building to have a special shape; we can add or remove chunks or rooms to create custom building styles; and we can adjust budgets for decorators and security. All of this means that adding a new city sector is easy, and we can even work towards personal styles for a particular corporation or building type.

So here are my two takeaways from this very brief algorithm rundown: it's okay if your generator makes mistakes and restarts a lot, especially if you're getting something good in return. And flexibility is **really** good - try and build your generators with lists, parameters, things you can tweak and control. The more handles you have on the weird thing you built, the better you can direct its weirdness towards the stuff you like the most.

Find out more about Rogue Process at www.rogueprocess.run



Procedural Generation as a Real-World Design Tool

By Paul Jeffries, Structural Engineer and Computational Design Lead, Ramboll
@pnjeffries

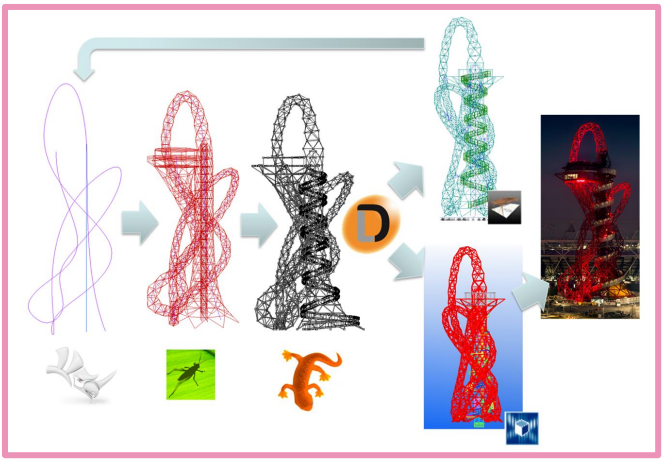
Procedural generation has been widely used in the creation of virtual worlds for games and movies, but it is also a topic that has significant applications in the design of buildings and environments in the physical world.

The field of 'Computational Design' is an emerging discipline in Architecture and Engineering that uses algorithms and computer code as tools in the design process. Instead of drawing out or modelling designs manually, computational designers instead create computer programs that procedurally generate the final design, based on a set of input parameters. This allows for the resolution of designs that would be far too complicated to create by hand and also enables the input parameters to be changed and optimised throughout the design process to produce the best possible solution.

This is a design approach with a long pedigree. The famous Spanish Catalan architect Antoni Gaudi applied a generative approach to some of his designs over a century ago, utilising hanging chains to model (upside-down) complex branching systems of vaults and arches that naturally adopted highly efficient structural forms. He controlled the lengths of the chains and the positions they were hung from, but the final geometry was generated by the force of gravity acting on this system and was, consequently, optimised to

"This forces the player to be fast and keep moving, or else risk falling off the platform."

resist those forces. These days we have the advantage that we are not restricted to the use of basic physical processes like gravity - with a computer we can model pretty much any process we can dream up – but the basic philosophy is the same.

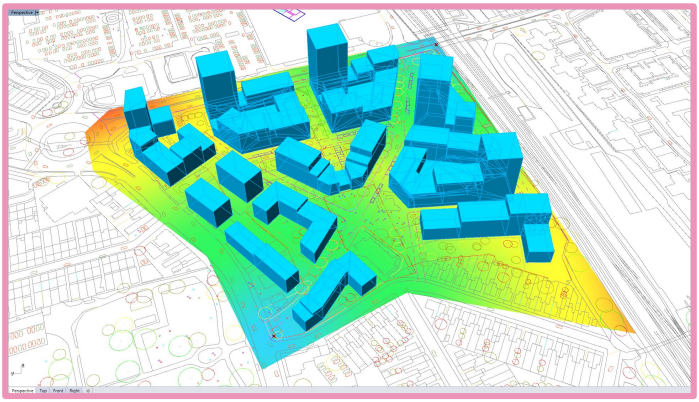


As an example from my own experience in structural design, the ArcelorMittal Orbit sculpture by Anish Kapoor produced for the London 2012 Olympic games has a highly complicated structure by most standards. But, by describing the geometry procedurally we were able to generate the whole structure from just a single spline curve, a couple of numeric parameters and a few straightforward rules. We could then control the whole geometry from just those inputs, making it easy to tweak the design and collaborate with the artist to produce an output that he was happy with and that we were confident would stand up.

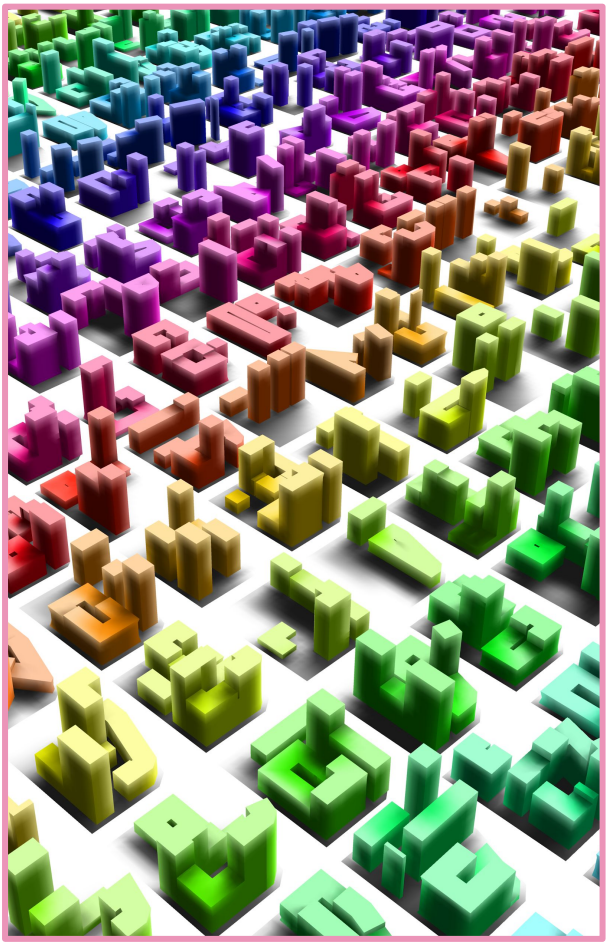
While originally a fairly niche approach used mainly on very unusually-shaped projects, the last decade has seen the rise of a new breed of node-based visual scripting tools that allow design geometries to be defined by connecting together pre-built and custom code modules. This has made these kinds of generative techniques far easier and more cost-effective and is rapidly pushing computational design more into the mainstream, being used in the design of everything from pavilions and small art pieces to skyscrapers, airports and sports stadia. It is likely that soon nearly all of the new constructions you see going up around you will have had some element of procedural generation involved in their design.

One major difference in the way in which procgen is used in architecture vs games is that while games leverage the randomisation of variables to generate a great variety of different outputs, building designers are ultimately interested in producing only one final structure and so the input parameters to the generative process are carefully controlled by the designer (it is often called 'Parametric Design' for this reason). However as part of an ongoing research project we call 'Dynamic Masterplanning' we are leveraging random number generation to apply a new 'blue sky' approach to building design.

At Ramboll, we use our engineering expertise to work with architects to design buildings and infrastructure projects that will be structurally viable, energy efficient and sustainable. The earlier we can get involved in a project the greater a positive impact we can have, as the overall arrangement of a building will have a major influence on how efficiently it can be constructed and operated. Typically, only two or three possible building arrangements will be considered in any detail in the design process, and often before engineering advice is sought.



This can lead to suboptimal designs and major setbacks later on when it is found to be impossible to make the building meet requirements. However, by utilising procedural generation it is instead possible to create thousands of different viable configurations for a particular site's constraints. By automating and integrating



engineering analysis into this process we can then test these different options and evaluate them through a variety of criteria. Many of the algorithms used in this will be familiar to games developers; we use ray tracing to check sightlines and evaluate daylighting throughout the year, pathfinding routines to determine shortest routes and travel distances across the site, voronoi diagrams to help assess loading distribution and so on. By doing this we can thoroughly explore the 'design space', trying out hundreds or thousands of potential options and allowing us to make better-informed design decisions based on a body of hard data.

A Conversation Between Botmakers: Part 2

By Heather Kelley & Audrey Moon
@PerfectPlum | @animalphase | http://www.perfectplum.com/ | http://loveme.computer/

Editor's Note: *This is a chat between botmakers Heather Kelley and Audrey Moon. In this second part of the interview, they discuss botmaking with Tracery, creating together, and exploring representations of the future. For the full interview, see http://www.procjam.com/seeds/*

[@perfectplum (PP)]: What wisdom could you give other bot makers who want to work with tracery SVGs?

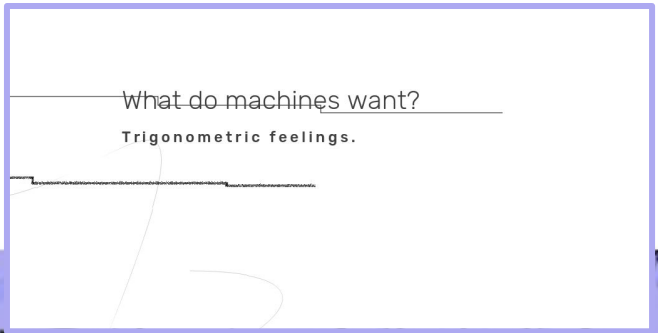
[@animalphase (AP)]: well, i know @v21 is working on CBDQ and fixing/improving things frequently, and it looks like @GalaxyKate is working on new versions of the language. my main tip would be, just play with tiny pieces of the SVG at a time to get a good sense of how the server will be happy to render your SVG. don't like, do it all locally and expect it to work when you paste it in!

"my main tip would be, just play with tiny pieces of the SVG at a time to get a good sense of how the server will be happy to render your SVG"

PP: hehe. word.

AP: keep talking to the system you're working with :) also, i'd say look a lot of code @v21 has done because there's a lot of really clever implementations for getting things like random numbers

PP: and in our case, reach out to the makers when it makes sense. i mean, i think everyone could do that, they are both very approachable



AP: yeah!

PP: like you actually did find some bugs!

AP: haha yeah! and v was super good to get right on things and fix them, or at least to investigate and offer insight for a workaround

PP: ❤

AP: it's good to work within a good community like this <3

PP: agreed, that was actually one of the main reasons i wanted to do a bot. there's the technical and artistic reasons, and the social reasons

AP: yeah, those two things are what keep me here in this space :)

PP: what were some of the creative challenges in interpreting the visual inspiration material (cybernetic serendipity exhibition) into an algorithm?

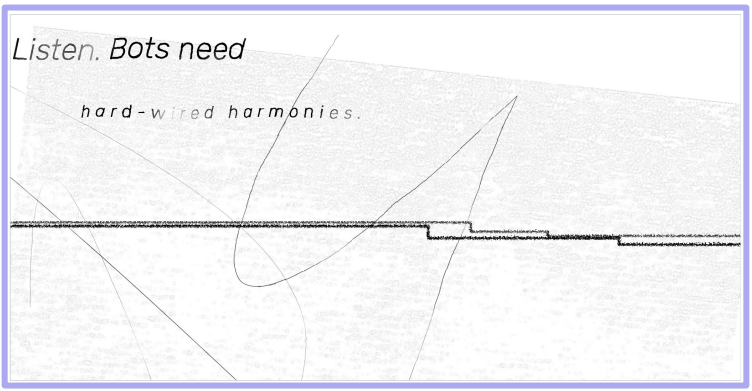
AP: ah! well, i think i was just trying to follow the same mindset of serendipity, which was a little bit nerve-wracking at first because we were working on a short timeline and that's not the best time to just play and see how things go lol. BUT the theme of the project was finding ways to let code- or machine-generated systems determine their own choices and paths in life, and the reference material was about that too. so, i had a bit list of elements to focus on, and kinda worked through those ideas with code implementations until we found something harmonious, something that frequently generated interesting results naturally from the code and that we could both look at and say "hell yeah"

"...the theme of the project was finding ways to let code- or machine-generated systems determine their own choices and paths in life..."

AP: so, that's still kind of that same mindset of creating something digital to appease our tastes, but, i was trying to find a bit of that balance of, playing with different kinds of code that tended to work a particular way, and find the bits of code that vibed with us for a mutually-agreeable relationship of harmony. but it was also done under a short timeline, so these decisions were also forced into a schedule that needed an answer fast! i think it's interesting because we came up with something that almost looks like documentation, so it's kind of like the digital processes are doing what they want, and what we get is the human-readable documentation translating their wants?

"The system cleverly knows how items work together and fit into quests, but the end result is... ridiculous."

AP: that's, kind of a feel i get from it. what were your thoughts assembling the references, and collaborating as we worked through it?



PP: well i was nerding out of course. it was easy on my side, just going down the rabbit hole, starting with the web site that someone is compiling with all the online references [1]. at first when we started i didn't have that exhibition in mind, and i can't remember when/why it occurred to me. But it's also about that feeling of being both "computer art' and very analog. since there was a lot of sculptures, and plotter stuff in that show. in fact stuff

that looks really contemporary now, with so many current creative coding projects meshing those old techniques with robots and AIs. And I also like the optimism of that time. When I told Golan about my topic, he made reference to an article that had just come out about how your sex bot could kill you. And while that's true on one hand, it's like, there's already a shit ton of people thinking about these dystopias, so few take any kind of hopeful stance now. And, without being a pollyanna, I wanted to at least represent that playful and hopeful side of things.

AP: yeah. also like your toaster could kill you and your car could kill you and, well, like you said we're gonna be surrounded by AI so i guess it'll be important to make friends with them!

PP: Yeeeeep.

AP: but yeah, i agree there's so much dystopia speculative fiction out there and to me it is more interesting and useful to focus on positive speculative fiction. like, "hmm maybe some things seem tricky now so what are some ideas about moving forward"

PP: Also, all these arguments still "other" the bots and act as if there is not some kind of equally dangerous/problematic human equivalent. It's again colonial and sexist. Only the most privileged of folks are worried about being murdered by their personal bot. Most of us are already worried a lot more about the humans we've already got all around us.

PP: So yeah, i want humanity to actually put some thought and action into how we might build a positive scenario! For the bots, and so for us too!

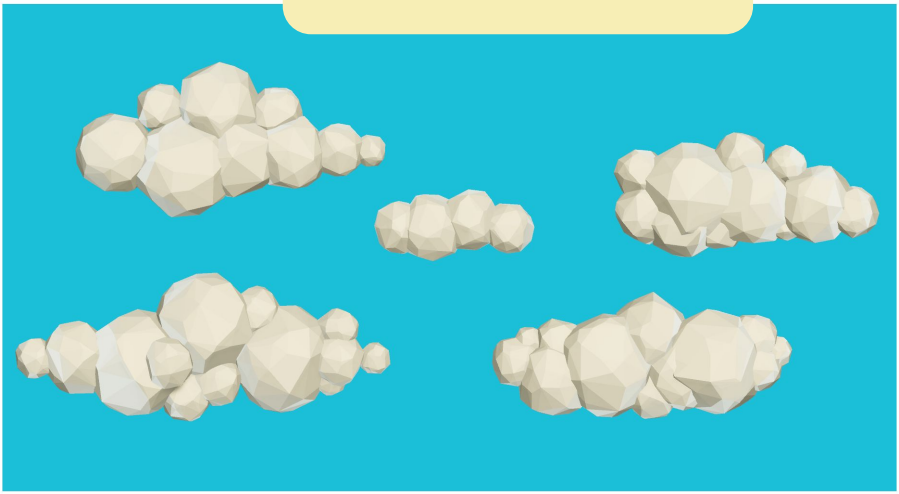
References

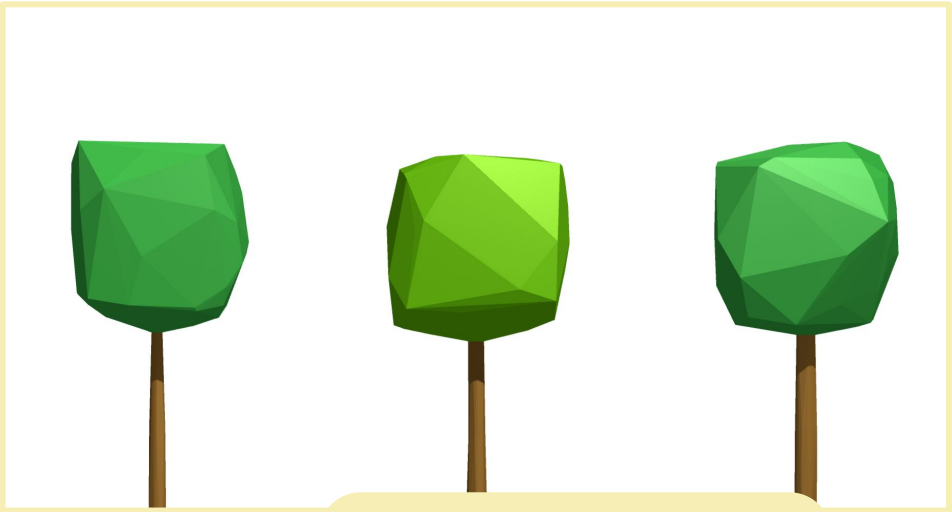
[1] http://cyberneticserendipity.net/

Procedural Low Poly

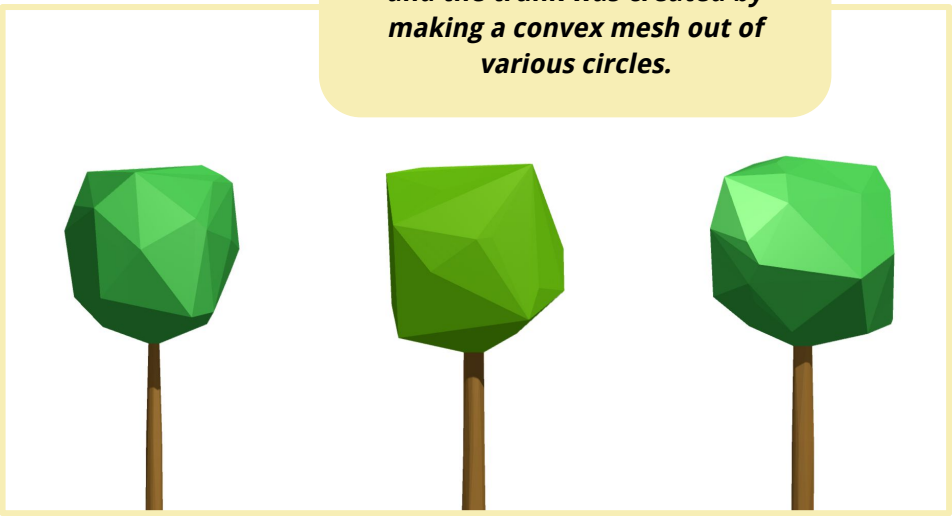
By Davide Prati of NoobStudios
@noob_studios | https://www.facebook.com/noobstudios/

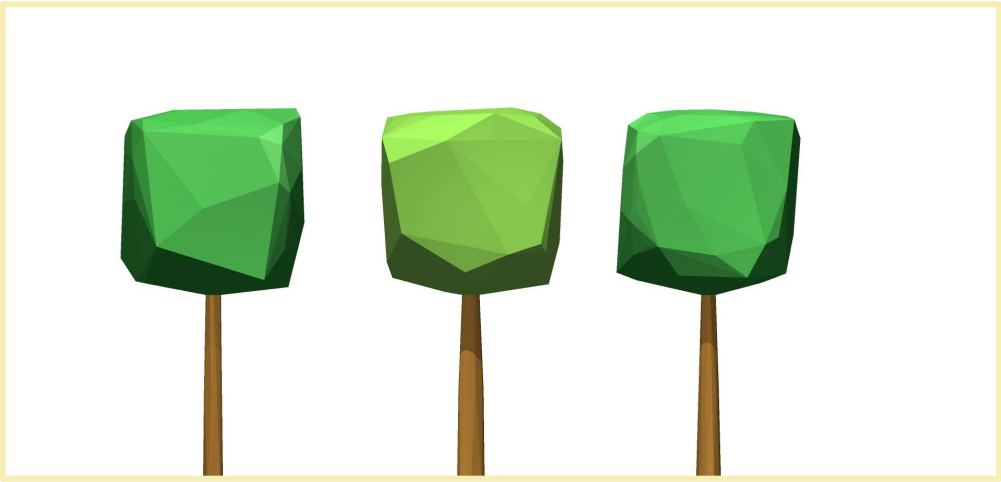
Sphere components that generated points inside of them and made convex meshes.

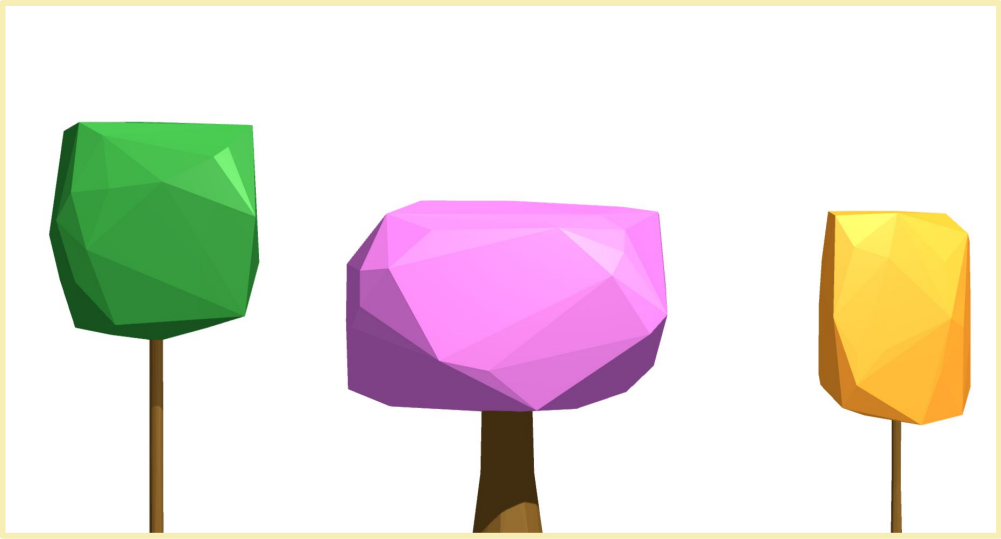


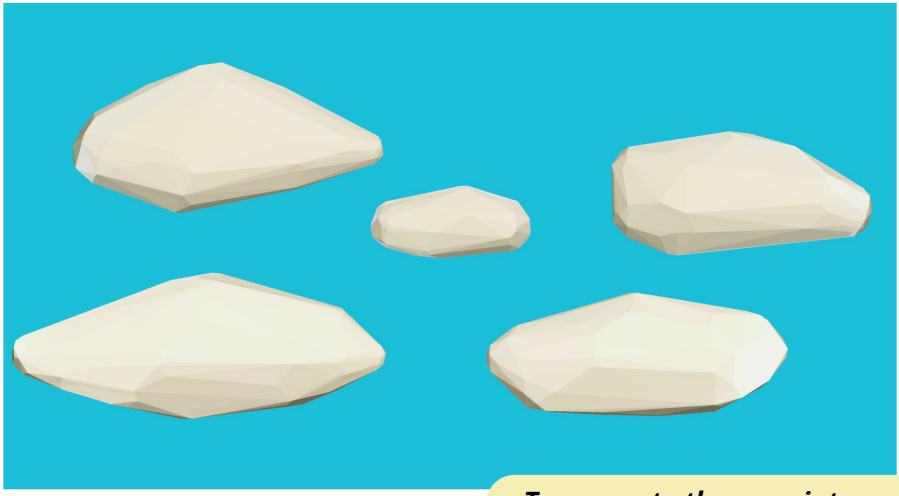


Leaves were made by generating a random convex hull inside a cube, and the trunk was created by making a convex mesh out of various circles.









To generate these, points were generated inside a cube and turned into a convex mesh that was then flattened.

Popular Visual Depictions of Early Generative Systems

By James Ryan
@xfoml

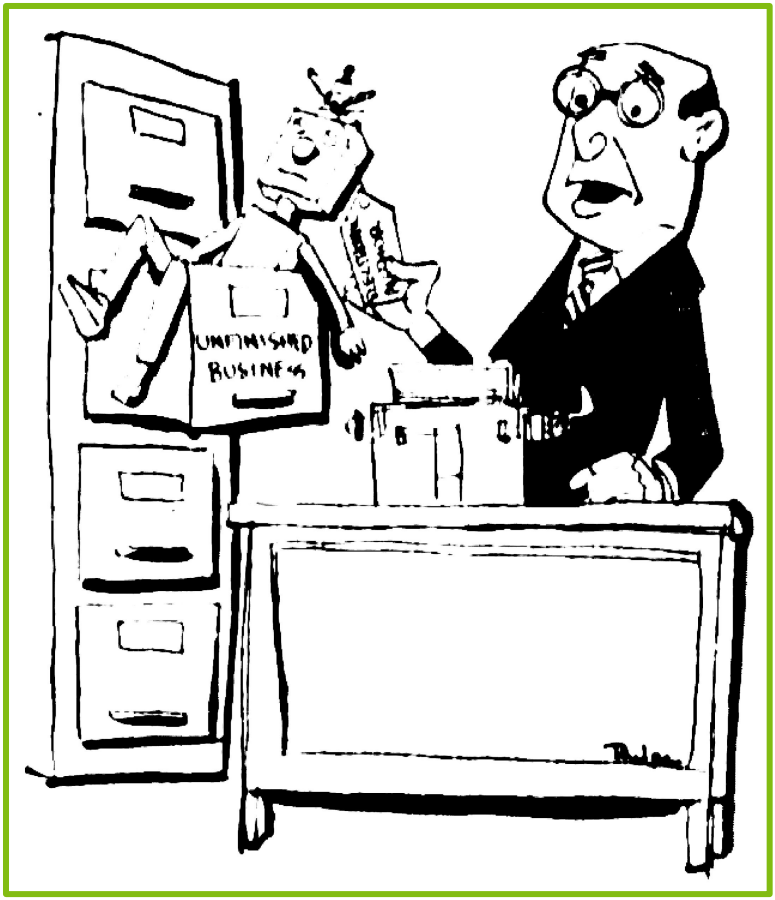
At its advent, the general-purpose computer was already being envisioned as a mechanism for procedural generation. Writing in 1843 about Charles Babbage's Analytical Engine, Ada Lovelace proposed the notion of computer music: "the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent". When the computer fully arrived the next century, Alan Turing promptly suggested, in 1949, the inevitability of computer poetry. The current earliest known project in procedural generation is Christopher Strachey's love-letter generator of 1952, and several more followed in the ensuing decade. Thus, procedural generation by computer is as old as the computer; procedural generation, more broadly construed, is probably as old as the procedure.

I spent the summer of 2017 excavating materials pertaining to the forgotten early history of expressive computation, which turned up, among many other things, a series of popular visual depictions of early generative systems. These illustrations appeared in newspapers, magazines, and other publications in accompaniment of sensational articles puzzling over the latest developments in procedural generation.

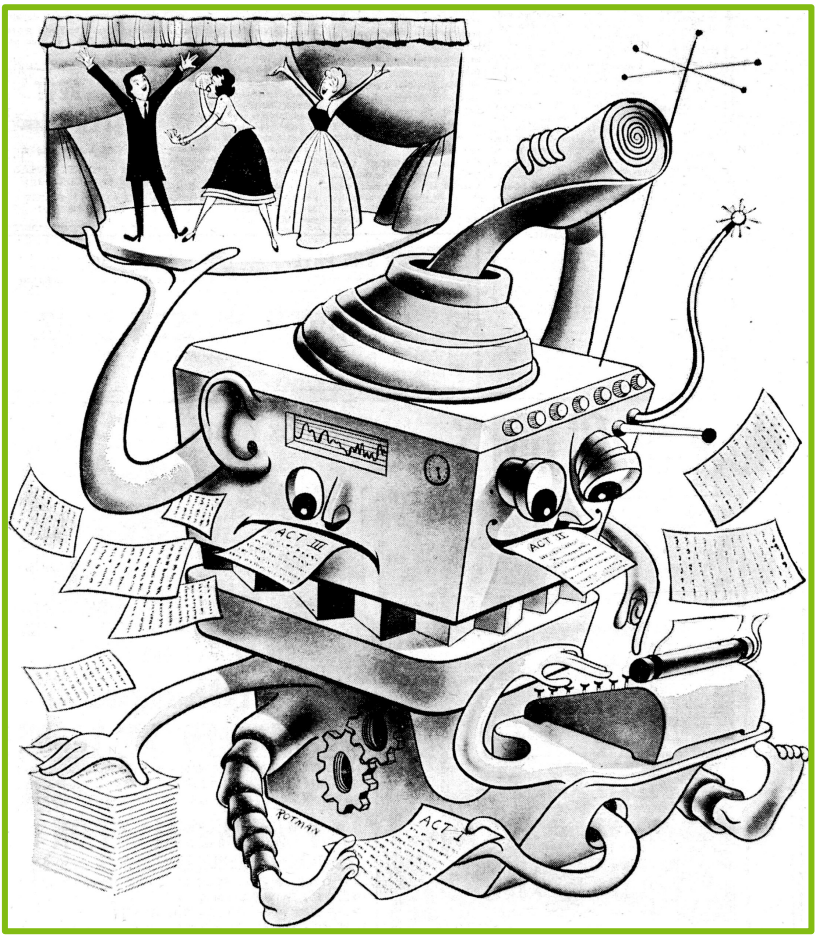
"For your enjoyment, I present here my collection of popular depictions of early procedural generation."

For your enjoyment, I present here my collection of popular depictions of early procedural generation. In all cases, these illustrations were published anonymously, though some are signed by the artists. Below, you can find brief descriptions of each image, along with citations for where I found them (though note that in some cases the images were published in multiple sources and likely originated in a press release or newswire package). Beyond the anthropomorphism, note the conflation of hardware and software that characterizes nearly all these images. This misconception was also present in the source articles containing the original illustrations, where generative computer programs were frequently referred to by the name of the computers on which they were

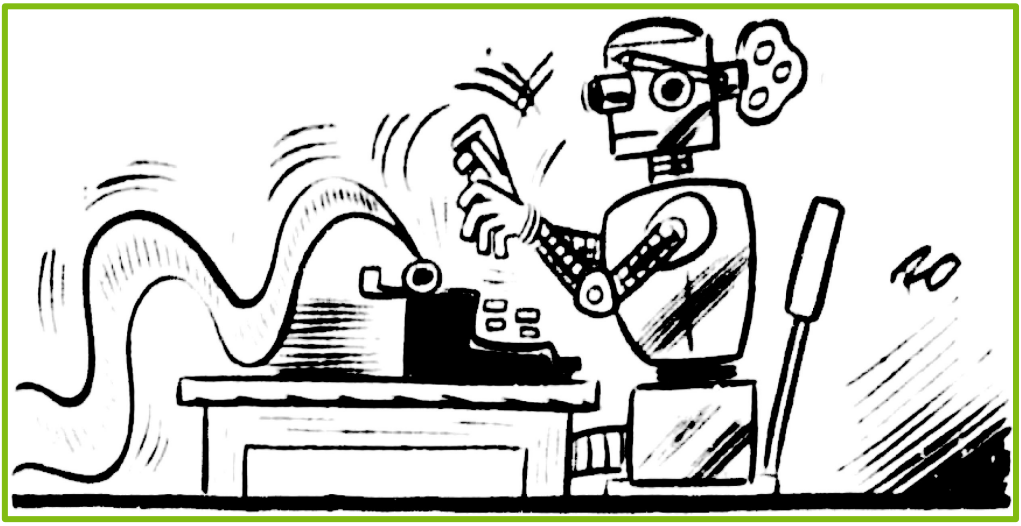
implemented (e.g., 'TX-0' for SAGA-II).



This image appeared in an article highlighting recent developments in procgen, including Strachey's love-letter generator and early computer music. The robot's tag reads 'electronic author', and it's being placed in a drawer marked 'unfinished business.' Story generation would emerge four years later, with SAGA-II. Source: The Indianapolis Star, Jan. 23, 1956, p. 10.



Here is a lovely portrait of SAGA-II, the first major project in procedural generation. This MIT system wrote screenplays in the TV-western genre and was featured on a CBS television program called Tomorrow. Source: Binghamton (New York) Press and Sun-Bulletin, Oct. 22, 1966, p. 15.



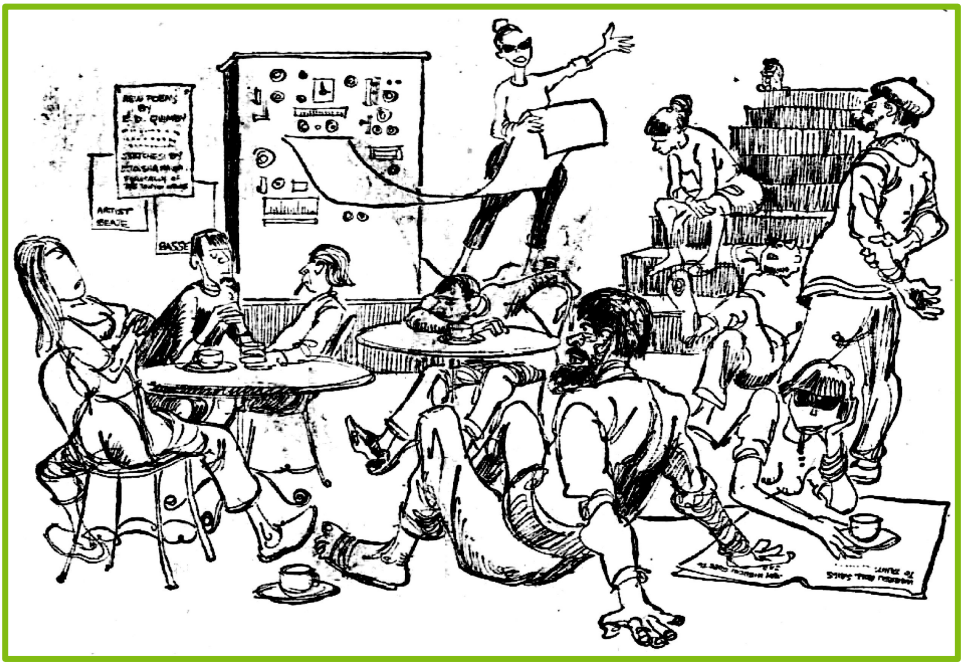
An alternative portrayal of the screenwriter SAGA-II positions it, curiously, at a typewriter. Source: The Pittsburgh Press, Oct. 26, 1960, p. 43.

Man and his "brain children."

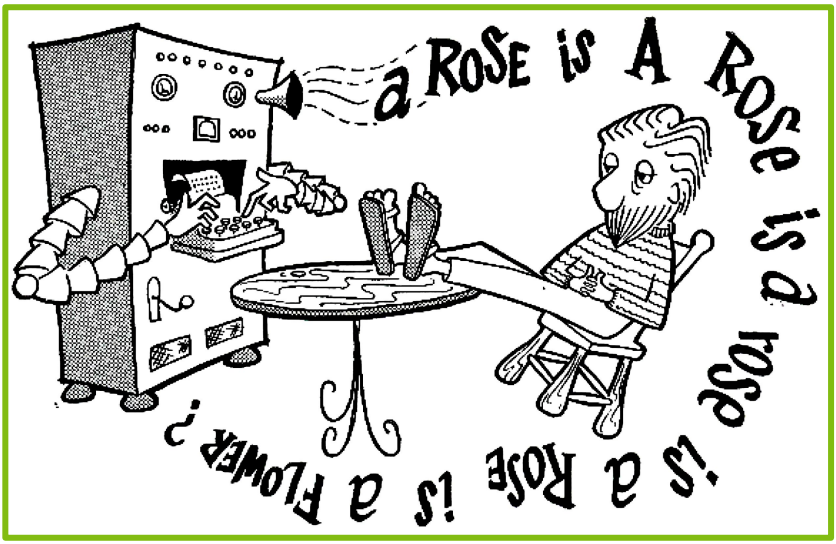
Rather than a depiction of an expressive system, this titled illustration captures the sheer processing power of computers, a feature that naturally captivated the mid-century imagination. Source: The Lincoln Star, Oct. 23, 1960, p. 48.

One of the first expressive systems to receive widespread media attention was AUTO-BEATNIK, an early computer poetry generator developed by engineers at the California-based Librascope company. This image depicts the smug computer poet in a publisher's lobby, waiting patiently to sell some of its copious outputs. Source: Binghamton (New York) Press and Sun-Bulletin, Feb. 4, 1962, p. 46.

Multiple contemporaneous sources claim that a Librascope engineer actually read some of AUTO-BEATNIK's generated poetry at the Venice West Cafe, a popular beat hangout in Venice, California. This image commemorates the event in a humorous way. Source: Binghamton (New York) Press and Sun-Bulletin, Feb. 4, 1962, p. 46.

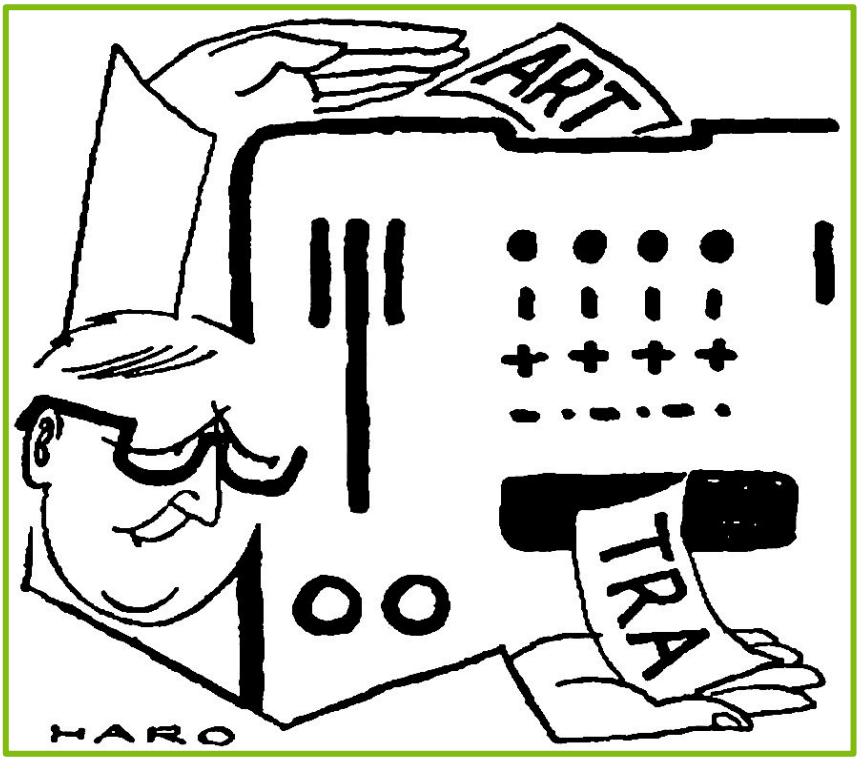


Here is another image depicting the public reading of AUTO-BEATNIK's poetry. Of course, the mainframe itself would not have been lugged along to Venice. Source: Honolulu Star-Bulletin, Apr. 22, 1962, p. 12.

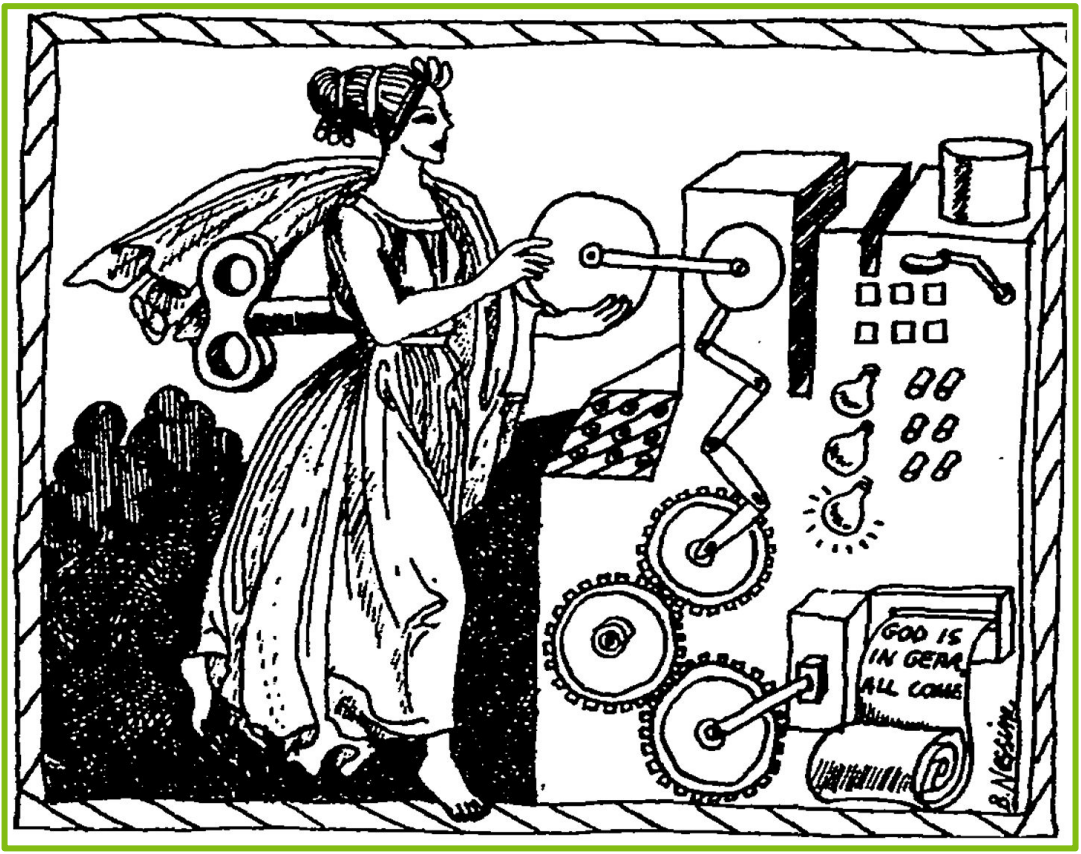


This illustration accompanied a faux interview with the computer poet that appeared in Librascope's company newsletter, the Librazette. Source: Librazette. July–August, 1962 (Vol. 9, No. 11), p. 11.

The inherently funny notion of a robot with writer's block is shown here. Source: The Sydney Morning Herald, Aug 5, 1962, p. 77.



This illustration appeared in an article about the increasingly real prospect of computer art. Source: The Observer, Apr. 21, 1963, p. 30.



Who then is the computer poet's muse? This New York Times illustration offers one captivating answer. Source: New York Times, Dec. 4, 1966, p. 63.

This striking illustration does not pertain to procedural generation or expression, but rather one supercomputer's sheer capacity for data. Incidentally, that machine, ILLIAC IV, was the descendant of the eponymous author of the most famous early work in computer music, "Illiac Suite" (1957). Source: ILLIAC IV (marketing leaflet), 1974, p. 9.