So you want to build a generator...

This is a beginner-level advice essay for people just getting started with building generators. It's also for practiced experts who want a way to organize their knowledge. The advice is meant for any kind of generators: humorous twitterbots, fanciful art-making bots, level generators, planet builders, makers of music, architecture, poetry, and cocktails. (edit: it turned out to be 4000 words long, so enjoy the ride!)

With so many possible kinds of generators, what advice can I give? Any advice will depend on what *kind* of generator you want to build. I'll give you a list of **questions** to ask yourself, and advice to match the possible answers.

The first question is: **What are you making?**

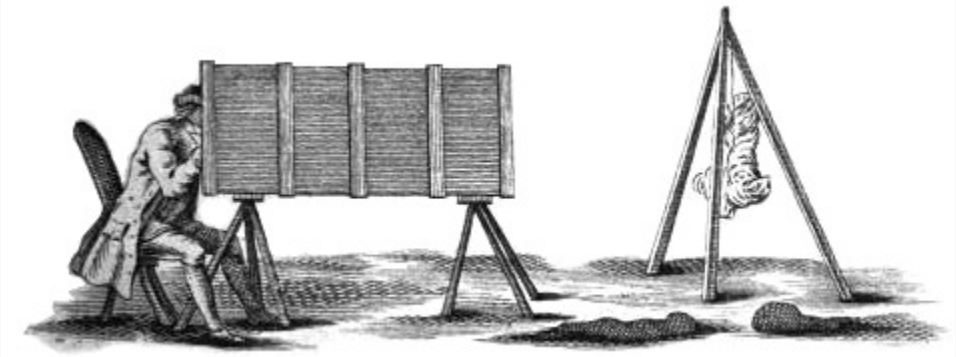
Write down the *thing* that your generator will make. We'll refer to this as your "artifact" but it could be anything: procedural birds, generated stories, animated dance choreography, gazpacho recipes, RPG quests, chess variants.

Now the hard part. Inhale, exhale, and start writing down everything you know about what makes one of those successful. What qualities does a good one of those *artifacts* have? "Funny", "Harmonic", "Playable"? Now go deeper: the more specific the kind of thing you are making, the more specific and detailed you can get with these requirements. What does a "fun" level mean? "Fun" will mean very different things for Super Mario or Civilization or Bejeweled. I can come up with more properties of a *good paranormal regency romance novel* than I can come up with rules for a *good novel*. **The easiest generators to make are the ones where you can describe "good" artifacts as sets concrete properties.**

Now flip the question on its head: what makes a "bad" example of this artifact? List everything you can think of, including properties that are merely unfortunate, and properties that are total deal-breakers. Put a star next to anything that absolutely *must not happen*. These are your *constraints*. **The most reliable generators are the ones where you can concretely describe constraints**.

You now have a list of desirable **properties** and **constraints** that you want and don't want for your artifacts. We want a generator with a *possibility space* (all the kinds of artifacts it can generate) where most of the artifacts have the good properties, and few (or none) of the artifacts have bad properties. We'll also want a *range* of artifacts, not just the same perfect artifact over and over and over, though how *wide* a range that is is a choice for you, the designer (a problem identified by Gillian Smith and being explored by Michael Cook).

Now we have the guidance we need to start building **methods** that make artifacts.



An alternative approach: the previously described method is good for inflexible situations, where you know ahead of time what you want to build. For many situations, like game jams or prototypes or side projects, you can be more flexible and improvisational! You can start with a methods or some loose pieces to figure out what it 'wants' to generate (what properties is it best at generating?) and then revising what artifacts you are generating to better suit what the generator is good at making. I talk more about this approach in my 2015 Procjam talk.

Building Your Artist-in-a-Box

When the Spore editors were being designed, the engineers, designers worked closely with the art team to understand *how* an artist or an animator would go about sculpting and texturing a creature. If they could understand the *process* and design an algorithm that could follow that process on demand, they would have an "Artist-in-a-box" that could make procedurally generated creatures that would be nearly as good as the original animator would have made. Designer Chaim Gingold explained this process in his 2007 GDC talk audio slides and Art Director Ocean Quigley used a similar process to experiment with what kinds of building blocks he would want for building cities

It is helpful when making a generator to sit down with someone who makes the sort of artifacts you are building, and have them walk you through the process of making something. What questions do they ask themselves along the way? How do they make decisions? How do they describe the tradeoffs between choices?

How do they describe the different problems that they have to keep in mind? How do they name all the parts of what they are working on, and all the relationships between them? *(the academic term for this is an "ontology")*

Some fields have expert practitioners who have written *frameworks*, often controversial, to describe what they do. Music theory has proposed plenty of rule systems, for example, for jazz improvisation or Bach-style harmonies or pop songs. Story generation has narratological theories like The Hero's Journey, but also informal frameworks like TV Tropes. Art theory has the Golden Ratio and color harmonies and composition rules (I haven't found those visual aesthetic rules to be productive in my own work, though you may feel differently). No framework is complete, or makes provable good artifacts, but they can help give you guidance and inspiration.

So now, ask yourself: **How would a human solve this problem?**

From rules to generative methods

Unfortunately, knowing how a human would make something isn't the same as being able to teach a computer how to do it. Humans are good at estimation, and making guesses, and synthesizing a lot of knowledge about past situations. Computers know only what you tell them, and many problems require way more implicit knowledge than we think, but they *are* good at performing lots of calculations and trying out lots of possibilities. So the methods we want to use will need to provide a way for the computer to solve problems *like* a human, or at least with a way to mirror some of the skills of a human. Methods that are particularly good for building generators (**generative methods**) will give the computer some of the following skills:

- Encapsulate knowledge of options (skill A)
- Create some structure (skill B)
- Encode conditional rules for options (A2)
- Create variability in structure (B2)
- Be able to ask itself questions about its constraints (have I solved this?) (skill C)

Distribution

This is the easiest kind of generative method. You have a bag of *stuff* and an area of space or time that you can spread them out across. Distribution methods usually don't have much overall structure (B), but they are often very sophisticated with the odds of picking each option (A). Some use weighted randomness to change the distribution percentages, or 'deck shuffling' (putting all the options in a stack and discarding when they are used), which prevents accidentally picking the same option twice. The conditional rules (A2) can get quite complex as well, but specifying *arbitrary* conditions is difficult to implement in practice. Most systems have carefully chosen parameters that can be set for each option and the conditional functions can just look compare the fixed parameters to make choices.

For an RPG example, wandering monsters are distributed around the environment (A). Higher monsters are found in higher level areas, water monsters in water, etc (A2). There may be a little structure to how they are distributed, like several 'baby' versions of a monster leading up to the 'boss' version. Loot is also distributed: you may be more likely to get high level loot in high level situations (A2) but there's still some randomly selected stuff chosen from a big list of possible loots (A).

Distribution in music and language doesn't work well. Randomly selected strings of words or musical notes don't have enough structure to make interesting meaning. For structure-heavy artifacts, you might want tile-based or grammar-based methods instead, or for something with very fixed structure and not much variability you could try the Parametric approach.

Parametric methods

You have a pretty well-built artifact already, and you know that you can *tweak* it a little. You have music, and you can transpose it up or down, make it louder or softer. You have a teapot, and you can make the spout curve out farther, you can make the body tall or short, thin or fat, and you can make the base wide or narrow. You have a set of hand-made alien creatures, and you can tweak their legs to be long, fat, curved, or flat-footed, and their bellies to be slender or barrel-shaped, and this changes their voice as well. This is how creatures in "No Man's Sky" are varied. This is a very reliable and controllable technology! For 3D

models, can often be encoded as a Maya animation channel, allowing it to blend with other animations (a Spore trick used in the rig-block animations). But your variability (A) is only along fixed one-dimensional numerical paths, there is no structure variability at all (B2). You can see something "new", but never something surprising.

A more sophisticated form of parametric methods uses other forms of input, and can generate new artifacts based not only numerical, but also point-based, path-based and graph-based input. When you draw a line in Photoshop with a tablet pen, your path becomes the input to an algorithm that renders a brushstroke, taking into account pressure, speed, and tilt as parameters at each point. The Spore creatures, likewise, used Metaballs, a geometry creation algorithm that could create smooth tubes along paths in 3D space. Other algorithms for filling spaces and paths are Voronoi patterns, Perlin/Simplex noise, triangulation algorithms, 3D extrusion or rotation, and the Diamond-square algorithm for fractal terrain. **These algorithms are particularly well-suited for interactive generators, because the user can provide the input parameters for the generator.**

Want more? Inconvergent.net has more intriguing specimens of these than you could possibly ever need, with open-source implementations.

However, though these algorithms are delightfully surprising, they can often be too uncontrollable to provide the constraint-satisfaction needed for gameplay or other highly-constrained artifacts.

Tile-based

Chop the problem up into modular, identically-sized slots. Have a number of different hand-made solutions that can fill in these slots. The artifacts being created are just differently-selected-or-ordered sets of pre-authored solutions. A familiar example of this is the board layout for games like Settlers of Catan and Betrayal at the House on the Hill (or Civilization for a digital example). The island, and the mansion, are constructed from the same tiles each time, but laid out differently, which changes the gameplay. Some of the earliest forms of generative content I've found are the Musikalisches Würfelspiel from 1750s and earlier, with which pianists could put together "tiles" (in this case: measures) of music to produce a playable waltz.

Tile-based methods are great for small-scale structure (B) because the insides of the tile are pre-authored, but they have no flexibility (B2) for small-scale structure for the same reason. Large scale structure is harder to control: it can be totally random. There can be more advanced constraints about which tiles can go next to other tiles but then you may need a complex algorithm to solve for compatible layouts of highly-constrained tiles ("The beach tile can go next to a jungle tile, but cannot be within two tiles of a river tile and....."). Individual tiles have a very rigid

encapsulation of possible options (A), because each possible tile has to be authored by a human. These systems don't have enough knowledge to come up with good new tiles by themselves. **Tile-based methods work for problems that can be broken-up into small chunks where internal structure matters, but that can still create interesting (but not constraint-breaking) behavior when combined in different orders.**

Interested in more analog content generation? Additional forms of boardgame and role-playing-game content generation can be found in this paper by Gillian Smith, and this exploration of comics by Chris Martens

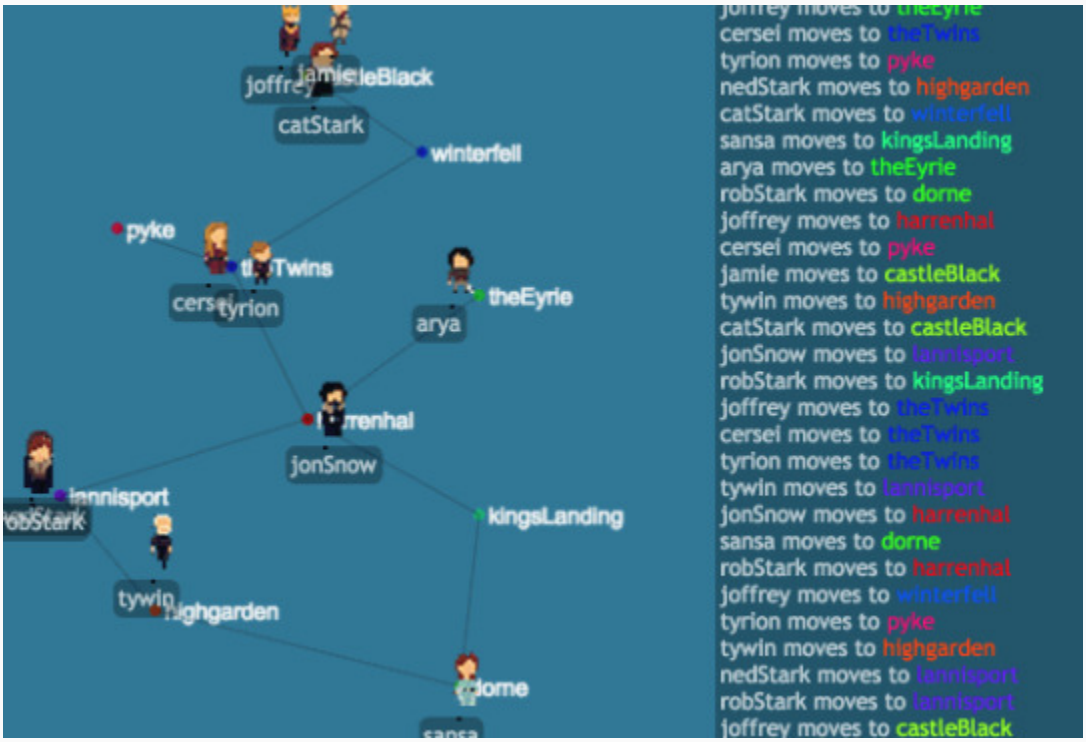
Grammars

Grammars are one of my favorite generative methods, because I find that they give me a great way to make very deep and complicated structures while still also having a lot of control over my options. Grammars are a computer-sciency way of saying that big complex things are made out of other things, and those other things may themselves be made out of even smaller simpler things. Orteil's

Nested is a lovely example of this. The universe is made of galaxies made of planets made of continents full of people who are full of thoughts and dreams and memories and atoms. Each symbol (that is, a type of thing) has a distribution of sub-symbols that it might be composed of. When it "unpacks" it has to pick one of those options (and any subsymbols then unpack recursively). **Grammars make it easy to encode knowledge about a particular artifact and its structure and its options all in one piece of data.** I like them so much I made a library to help people work with them, Tracery. This has been used to make a twitterbot hosting service and lots of great weird creative twitterbots including some of my own.

The downside of grammars is that they do not have a way to handle constraints, unless the constraints are implicitly encoded in the grammars themselves (if a bed can only be in a house, than only the house can have a bed as a sub-child, for example). It's harder for grammars to encode high level relationships between different things generated at different points in the grammar. If you wanted to have foreshadowing of the villain's death at the very beginning of the grammar, that might be difficult to do, and you might want to use the 'big hammer' of a constraint-solver.

Constraint solvers



Constraint solvers are very powerful, very recent tools in our generative arsenal. They are what you use when you have a lot of hard constraints, a lot of flexible and complex structure, but you don't know how to build out the structure in a way that will be *sure* to solve your constraints.

The oldest simplest version of this is **brute force solving**. Make *every* possible variant of content, toggle every switch, make an alternative universe where you have made each different decision, and test your constraints until you find one that works. This is a viable solution for some problems! But as any mathematician will tell you, too many choices will create a number of possible

artifacts to search that is greater than the number of atoms in the universe. And that's going to be slow.

There are often shortcuts you can take, depending on how your constraints are structured (I don't have to choose an ice-cream flavor in possible futures where I did not go out for ice cream). But this takes a long time to author by hand (just ask the Storyteller developer).

Fortunately, many mathematicians and logicians find it amusing to try to solve this problem, and they have come up with *general purpose solvers*. Plug in your constraints and structure and options (in a language that it can understand) and it will find all the cheap shortcuts to cut the aeons of brute-force solver time down to something slow but accomplishable within normal gameplay time.

Because these tools are big and bulky and new, they are still hard to plug into many game engines, and there aren't many tutorials. Adam Smith has been doing good educational outreach for Answer Set Solving, a particularly powerful one. Another one with some support is Craft by Ian Horswill, a constrained random number generator, that has been recently ported to javascript. **Look for these rare but powerful methods to be more common in the future!**

Agents and Simulations

This is where it gets weird. Remember how I said that we could look at how humans solve problems to provide inspiration for our generators?

Guess what: humans aren't the only ones who solve problems

There are algorithms that solve problems based on the colonial behaviors of ants, or the social communications of fireflies. Many other agents and simulations take inspiration from other parts of nature, like flocks of birds, evolution, bacteria, neurons and cities. Here are a few of my favorites, but there are many more.

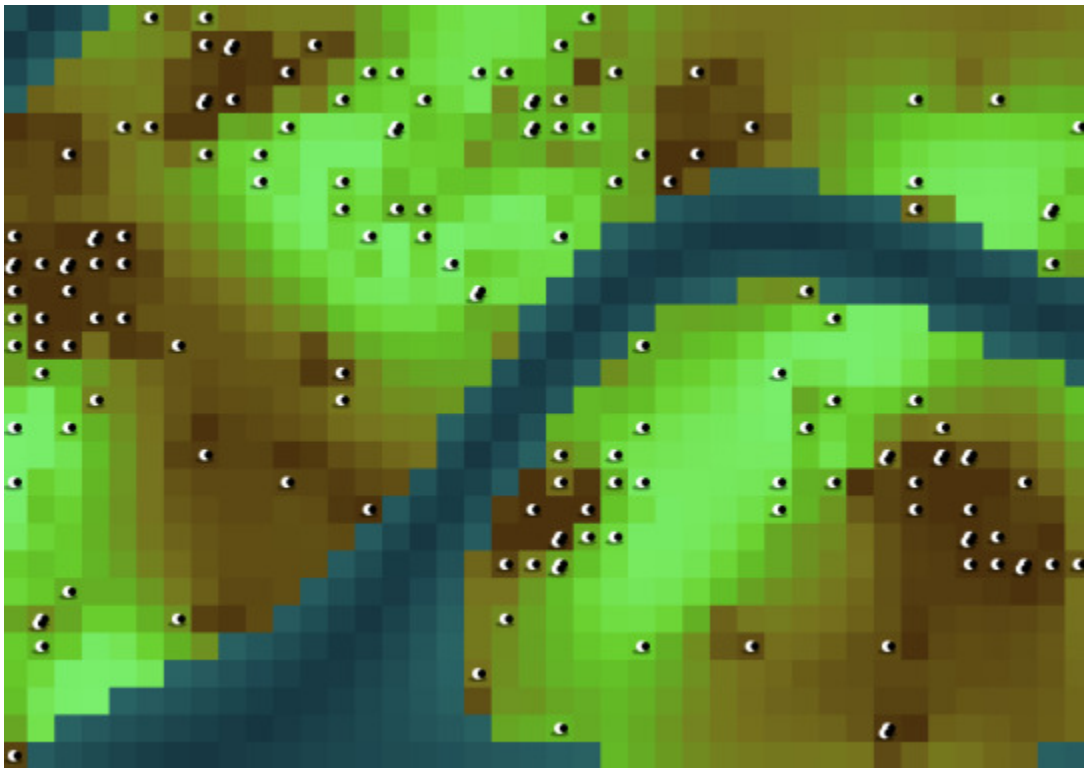
Steering behaviors (these deserve their own long post some day) can create remarkably complex crowd motion. Braitenberg vehicles were originally a thought experiment about simple machines with two photosensitive eyes and two wheels that could 'steer' themselves just by activating one wheel more than the other, applying an asymmetrical force that changes their direction. Despite their total brainlessness, they can show 'fear' and 'adventure' and have been ported to many languages and physical robots.

Boids took the steering approach of the Braitenberg vehicles and applied it to flocks of birds and fish (and the wildebeest in the Lion King animated movie). Each bird helps keeps its flock in shape by calculating applying its own forces for cohesion, alignment and separation. Small variations in the each birds's tuning values can generate new behavior for the flock.

I've also used steering forces to generate procedural dance: you just replace the flocking forces with rhythmic forces in time to the music paper. Steering forces can do a lot more than pathfinding, but I don't think they haven't yet been explored to their full potential.

Genetic algorithms, as can be seen in my flower-evolving app aren't methods to generate content, you still need a generator (a flower generator in this case). But they are a way to *guide* that generator towards more constraint-fulfilling and desirable-property-producing content. A genetic algorithm (which, yes, needs its own post), needs three things:

- A thing you can modify (a 'genotype')
- A thing you can judge (a 'phenotype')
- A way to turn the first into the second



For the flowers, the genotype is an array of floats. That array gets fed into a parametric generator to create a pretty animated flower (turning genotype to phenotype). The user can see which flowers they like (a judgable phenotype). They pick their favorite, it's original genotype is cloned and mutated (a thing you can modify, remember?) and the next generation is born from its mutant babies, and overtime, evolution happens. Sexual reproduction is common in GAs, but so

are many other interesting kinds of reproductions and metaheuristics. There's a lot of neat research in this area!

Cellular automata rely on many very simple agents, all working in parallel. The canonical example of this is Conway's Game of Life, in which many tiny automata in a grid, each with a very simple behavior, can give rise to so many strange behaviors and phenomena that many mathematicians make a hobby of discovering and cataloging them.

Cellular automata with more complex rules are used to create Dwarf Fortress, the classic Powder game, and with the advent of voxel-based simulation, are taking on a new life as the engine behind Minecraft

After you generate...

You've browsed this list of generative methods, you've got your list of constraints and properties in hand, and you've built your generator!

Now what?

Ways that generators fail

Something has gone horribly wrong. The content looks ugly. The content all looks the same. The content looks like genitalia. The content is broken. Some of these problems are easier to solve than others. Here are a few kinds of difficult problems you will encounter.

One will be the kind where you can *computationally identify* when something is going wrong. The solution to this one is to generate some new content until this constraint is no longer violated. Perhaps you want a generated chair to have its center of gravity over its support points (like legs), so it won't fall over. This is possible to calculate with a physics simulation, so if the generated chair fails, generate a new one until it passes. This approach is called "generate and test".

What can go wrong with "generate and test": what if every chair you generate fails this test? Perhaps content that passes the test is very rare. Or there are too many constraints, if you have constraints for material thickness and cost and symmetry and comfort and more? Each chair might satisfy most constraints, but with enough constraints, most chairs will still fail one or two. Maybe you need a constraint solver. Or maybe you need to restrict your generator so that it is more conservative with its choices, though that may lose interesting possibility space.

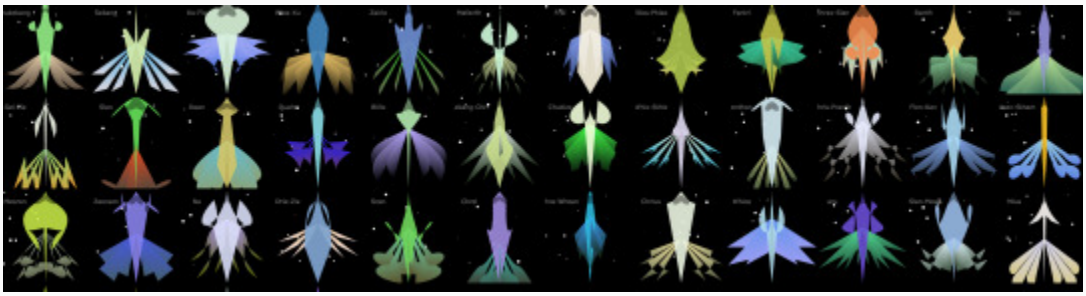
Another difficult constraint is when you cannot adequately describe your constraints. This is a remarkably common situation, because there are so many things that we don't want but can't write rules that "I know it when I see it" can be used as a serious legal argument.

Is this an offensive adjective to use to describe this character? Does this color palette look too much like a corporate logo? Does this look like genitalia? Is this just... ugly? This is a hard and unsolved problem, I'm afraid. If you can't define "bad" content, it becomes impossible to filter, especially when your human users are trying to get around the detection algorithm. In this case, the best path is to construct an generator such that it is *harder* or less-likely to make offensive content. This also restricts your possibility space, like removing words that are harmless and useful unless combined in a particular way.

Aesthetics: the toughest challenge

The most common way that generators fail is that they produce content that fails to be interesting. What is "interesting"? That depends on the situation. Very few generators produce only one of a thing. Most generate multiples, but a twitterbot posting every hour will generate more content than a novel-generator outputing one novel every NaNoGenMo. So achieving novelty with the first Twitterbot will be more difficult because there are so many artifacts being produced that any

given one of them will probably start seeming less special.



So your algorithm may generate 18,446,744,073,709,551,616 planets. They may each be subtly different, but as they player is exploring them rapidly, will they be *perceived as different*? I like to call this problem the **10,000 Bowls of Oatmeal** problem. I can easily generate 10,000 bowls of plain oatmeal, with each oat being in a different position and different orientation, and *mathematically speaking* they will all be completely unique. But the user will likely just see *a lot of oatmeal*. **Perceptual uniqueness** is the real metric, and it's darn tough.

In some situations, just **perceptual differentiation** is enough, and an easier bar to clear. Perceptual differentiation is the feeling that this piece of content is not identical to the last. A user glancing at a line of trees can tell if they are identical, or if they are less-varied-than-expected suggesting unnaturalness. This fulfills an aesthetic need even if no tree is particularly memorable.

Perceptual uniqueness is much more difficult. It is the difference between being an actor being a face in a crowd scene and a *character* that is memorable. Does each artifact have a distinct personality? That may be too much to ask, and too

many for any user to remember distinctly. Not everyone can be a main character. Instead many artifacts can drab background noise, highlighting the few *characterful* artifacts.

Characterful artifacts is another blog post for another time, but certain aesthetic principles create objects with *readable* meanings for human perception. Humans seem to like perceiving evidence of process and forces, like the pushed up soil at the base of a tree, or the grass growing in the shelter of a gravestone. These structurally-generated subtleties suggest to us that there is an alive world behind this object. Kevin Lynch's influential "Image of the City" demonstrates that there are factors that make cities memorable and describable. Perhaps there are other aesthetic rules that we can discover, too.

Conclusions:

Oh, my, this turned into a monster of a blog post. It's probably full of errors and omissions, but I hope it will be of use to you.

If you want to begin playing with generativity right away, here are some great tools to get your feet wet:

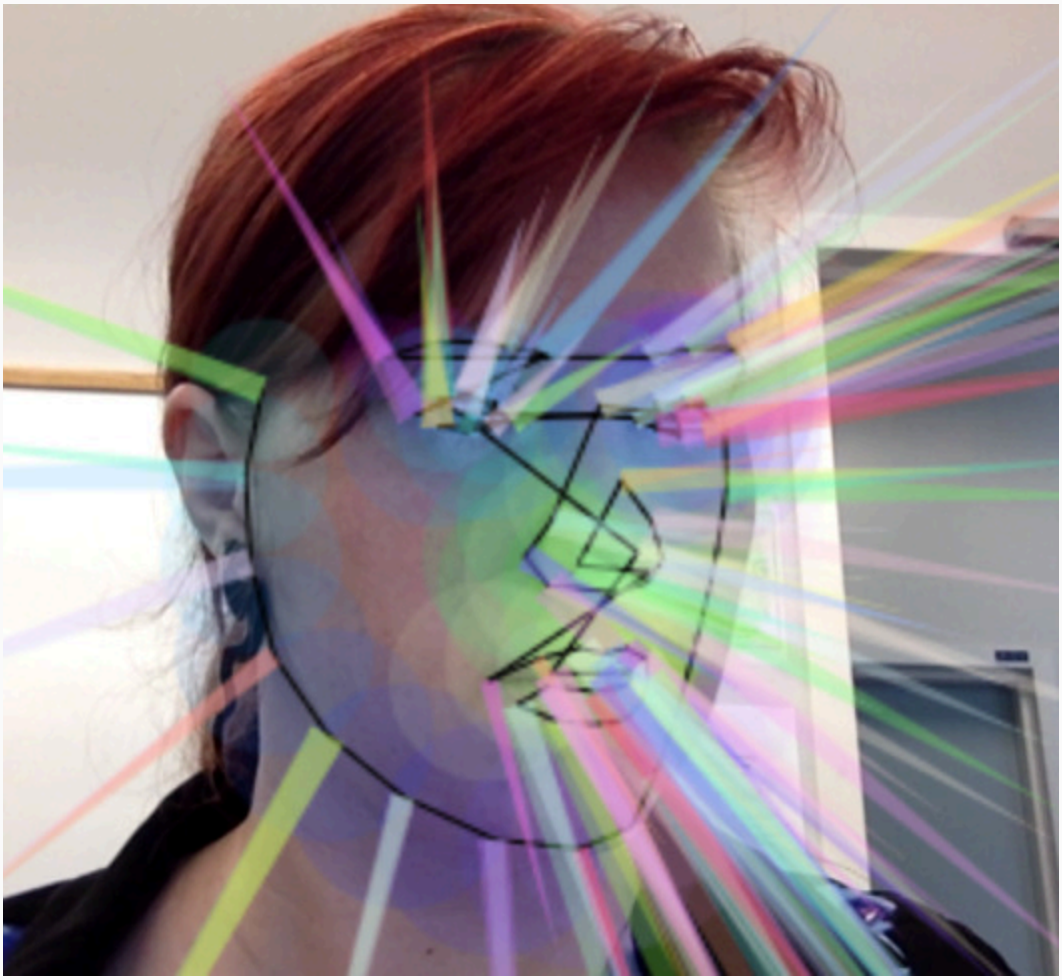
- http://cheapbotsdonequick.com/: the Tracery twitterbot maker, now with SVG supprt
- http://www.contextfreeart.org/
- http://ncase.me/simulating/ Simulating the World in Emoji, a brilliant playground for emoji-based cellular automata

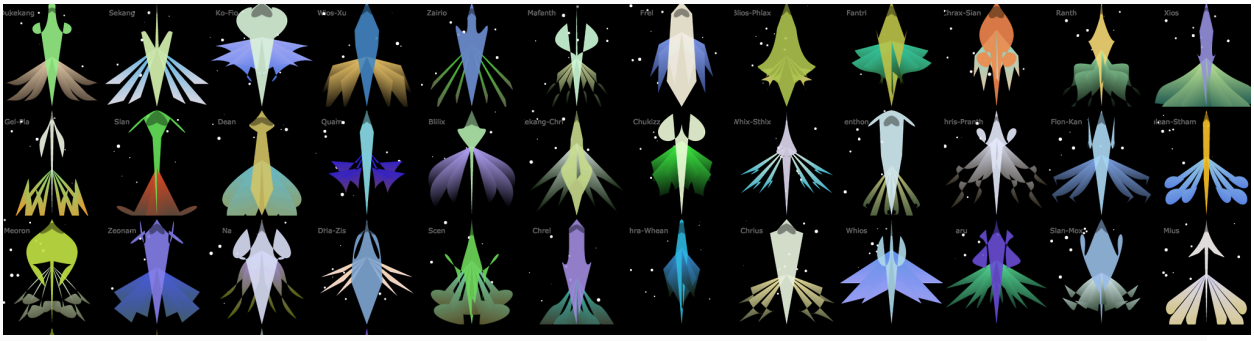
Want to learn more? Or meet like-minded people?

- There's a free online PCG textbook written by the top academic researchers in generation for games
- I'm absolutely in love with this book by the brilliant Daniel Shiffman that teaches Processing through the lens of nature's processes
- The Computational Creativity Group at Goldsmiths does extremely varied work in computational creativity, the practice of trying to make computers "creative". The International Conference on Computational Creativity is a place where many of these people meet to share ideas. Their past papers are a great resource on generating art, poetry, music, memes, and more!
- The Expressive Intelligence Studio is my home at UCSC, where we work on AI for *expressive* purposes, not just problem solving.
- Foundations of Digital Games is a yearly conference on experimental ideas about games, and has a Procedural Content workshop (and a new Crafts in Games workshop, too)
- Superstar academic/gamemaker Michael Cook runs the ProcJam, an annual jam that features lectures and tutorials and tools.

- Darius Kazemi runs the National Novel Generating Month, a challenge to make a program that can write a whole book. He also write a ton of bots, and writes a ton about bots

I'll be on Twitter, @galaxykate, to answer further questions, hear about typos, and take feedback





@galaxykate0 / galaxykate0.tumblr.com