

Seeds

Issue 4



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

Contributors:

Tanya X. Short
Shannon Kao
Neel Shivdasani
Justas Dabrila
Mikhail Maksimov
Alexander Zhuravlev
Jesse M. Porch
Hodge
Elle Sullivan
Joseph Alexander Brown
Hamna Aslam
Nikita Lozhnikov

Isaac Karth
Max Kreminski
Serin Delaunay
Lee Tusman
Sean Butler
Gordey Chernyy
Maria Mishurenko
Lee Tusman
Terry Trowbridge
Neil Bickford
Kelson Smith
Brandon Campbell

Organisers:

Mike Cook, Jupiter Hadley,
Azalea Raad, Dann Sullivan

**Thanks to the Royal Academy of
Engineering for additional funding and
support**

Discord Community

Managers: Hectate, KaynSD,
Stella Mazeika

Cover Art by: Martin O'Leary

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

<https://twitter.com/mewo2sketches>



ProcJam

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.

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

We hope you enjoy it!

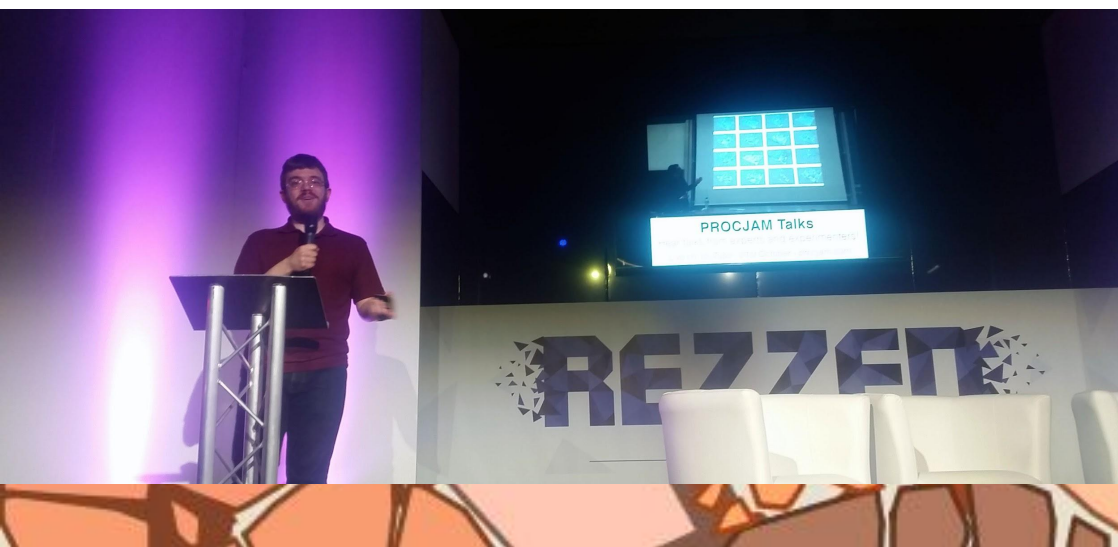


Table of Contents

Do's and Don'ts of Procedural Flirting	01
The Divine, in Verse	04
How GASP Gallery enables generative artists	09
Unbounded Prefab Based Dungeon Generator for Tiled Game Worlds	11
Infinite Graveyard Generator	16
Procedural Real Scale Planets in Real-Time	19
Narrative Deckbuilding for Ambient Storytelling	25
The Divine, in Verse	28
Writing programming languages for procedural generation	34
Paper Terrain Generation	38
Modeling Possibility Spaces in Graph as a Tool for Generating Detailed Worlds	41
Generators That Read	46
Incorrect Tracery	49
Throwing Things	54
Procedural characters for VR action game: 5 lessons learned	60
Paintmaster 5054 - Jazzfunk Greats	63
Report on the First #InnopolisAI Art Contest	65
Does Generative Poetry Need A Theory of Forgetting?	69

Do's and Don'ts of Procedural Flirting

By Tanya X. Short
@tanyaxshort

"However, dating and love are fraught with tricky social cues and political implications, so we knew procedural humor is a bit risky."

BadCupid was briefly a livestream-only game created for Mixer (perhaps someday to come to Twitch.tv) in which viewers could bet virtual currency on the outcomes of "dates" held between randomly chosen characters in a database. Characters would flirt with one another using different 'Moves', such as Compliment or Kiss, with accompanying text as dialogue or description of their action.

As a result of a Move and its success or failure, characters gain or lose Love points. Players bet on whether or not the two characters can reach maximum Love within a time limit.

When designing and scripting the A.I. characters "fall in love" in BadCupid, we intended to create a spectacle that was fun to bet on, like a sports match or horse race. Thus, our design goals were to: Make it funny, using subverted familiarity, surprise, and some absurdity.

Make it dramatic, with generally perceived/clear progression and outcomes, with over-the-top writing and "swingy"-feeling pacing, leading to somewhat predictable odds with occasional improbable, sudden upsets.

However, dating and love are fraught with tricky social cues and political implications, so we knew procedural humor is a bit risky. We wanted to create an experience that equally celebrated all forms of adult, consensual dating, and feared making light of sexual harassment or predation, which are real dangers in the modern



dating world.

In order to reduce the chance of unpredictably offensive content, we decided to use a “replacement grammar” for the dialogue, meaning it was all created from an authored corpus. So if a line’s base grammar was “%thinkPhrase %interest %positivePhrase”, for example, it could be filled to become “I’m sure science is so great.”

OR “I think Street Fighter should be more popular.”, but all of those variants (such as thinkPhrase becoming “I’m sure” or “I think”) would need to be written by us.



With all that in mind as context, here’s a few lessons learned from writing a replacement grammar for the flirting dialogue of BadCupid.

Do’s

Do be inclusive. It was tempting to limit ourselves to only ‘safe’, status quo depictions of romance, because we were afraid of depicting LGBTQ+ dates, or different body types, or other cultures, seeming like a joke. But by including lots of different characters with different known romantic preferences, it welcomed more kinds of viewers and gave the algorithm a seemingly wider range of outcomes, because we naturally

interpret some flirts differently between different types of people.

Do plan out your pacing. Escalation is the key appeal (and danger) of dating. To go from being strangers to falling in love is a pretty intense escalation – it can help to map out exactly what kinds of flirting

belong to which level of intimacy and extroversion one or both flirts have achieved.

Do radically accommodate character personality traits. One of our biggest mistakes was trying to make a single set of Moves support most characters -- it made them all sound too similar, and react too similarly to each other, when in reality, the fun and horror of dating is that we're all so different, even when given the same stimulus. One person's casual "Hello, beautiful" is unspeakable to another, and might be the perfect line on some of us, and the worst line on others. As soon as we started splitting up characters into at least "kind" and "jerk" types, we started seeing much more interesting flirt scripts.

Do highlight when characters use or discover a shared, mutual interest (or a new incompatibility). This kind of moment is the building block of any romance (or heartbreak).



Don't

Don't overly depend on "compatibility". In reality, humans are often picky and have specific demands of our dating partners (gender expression, age, location, behaviour, personality, etc), but flirting is much more compelling when you know there's a serious chance they'll fall in love, no matter how ill-suited to each other these 2 characters are.

Don't forget that flirting is a two-way, collaborative activity, not just 1 person acting on another. Spend plenty of time thinking and planning not just how characters can flirt, but also about the different ways characters can react to being flirted with!

Don't be afraid to go a little extreme, especially if your pacing plan can support it. At first, we only explored Moves like "Kiss" or "Touch" or "Insult", but if you would like your procedural flirting to be funny, why not go all the way with "Propose Marriage" or "Declare War"? It certainly makes the dates memorable in a short period of time!

But maybe most of all, don't hesitate to make a procedural flirting engine! The world needs more interesting deconstructions of what is romance and why we fall in love.

The Divine, in Verse

By Shannon Kao

<https://www.shannonkao.com> | @shannonkao

PROLOGUE

「Oaks」

In a sweet unrest, tall oaks, branch-charmed by
the earnest stars,

「Wakeful」

Beginning, and the wakeful bird; your place is
changed; you art the same.

「Planet」


Fountains' interchanging kisses, when a new
planet swims into his ken;

Friends at the Table is "an actual-play podcast* about critical worldbuilding†, smart characterization‡, and fun interaction between good friends", run by the inimitable @austin_walker.

* A podcast recording of a tabletop roleplaying campaign

† of which the fourth season--Twilight Mirage--is a mecha anime story set in a far-future utopian fleet of city-ships

‡ each associated with a synthetic intelligence called a Divine.



Divines are massive mechs, and work in concert with a pilot, called an Excerpt.

This is not critical to understanding the generative system that follows, but Friends at the Table is a very good podcast. The following_is_critical:

Excerpts, as the name might imply, are named after a fragment of a text, with one word annotated for abbreviated address. Examples from the show include:

- _With The Fourth Promise Broken, The People Of The Clay City Watched The Sunset For A Final Time_, known as 'Promise'
- _They marked scars of light in pitch; born in fiercest purpose, and beheld as the signet sealed upon our pact_, known as 'Signet'
- _Mighty though the force of the tides be, the swimmer's will extends further yet_, known as 'Will'

I'm a longtime fan of Friends at the Table and their superlative naming systems, and was inspired by @longestsigh's Friends at the Table NPC name generator* to make my own, specifically for these excerpt-style names.

* <https://twitter.com/longestsigh/status/1145098049833897986>

PROCESS

「Breath」


one faint breath can enter here -- that holy light
is cast.

My initial idea was to find a repository of interesting prose and parse sentence fragments from it. I poked around Wikipedia's random article APIs, thinking to pull from a category like "Music" or "Poetry", scrape the text, and parse it into something resembling an excerpt name. Ten minutes of searching didn't reveal an easy random-article-in-category API, but DID inspire me to see if there were pre-existing databases of poetry I could query, wherein I immediately discovered PoetryDB.org.

This incredibly convenient and fairly extensive API allows you to query poems using author, line count, title, or the content of a line itself. A request for all lines of Emily Dickinson's poetry returns:

```
[
  {
    "lines": [
      "Not at Home to Callers",
      "Says the Naked Tree --",
      "Bonnet due in April --",
      "Wishing you Good Day --"
    ]
  },
  {
    "lines": [
      "Defrauded I a Butterfly --",
      "The lawful Heir -- for Thee --"
    ]
  },
  {...}
]
```

My first cobbled-together attempt randomly picked either Dickinson, Shakespeare, or Tennyson, concatenated two lines from the corpus of



that author's work, and grabbed a random chunk of words from the middle of that string. Keeping both lines from the same author, I figured, would ensure some continuity of style and theme. I then "modernized" the string with a quick search-and-replace of things like "thou" with "you", and picked a random "interesting" word from the name to use as the shorthand character "nickname". (Interesting-ness here determined by an extensive blacklist of boring words. Like prepositions.)

「Orchards.」

the fadeless orchards. The Moon upon her fluent Route

「Scythe,」

sun blaze on the turning scythe, She answered, 'but it pleased
us not: in truth

There are some fiddly issues with punctuation, specifically the join between the two lines, but this first draft worked astonishingly better than I had anticipated. To be honest, it feels like cheating to take any credit when writers of this caliber did most of the composing, but I was completely delighted by how interesting and coherent the initial results were.

I did some additional work to polish up the string parsing--randomized punctuation between the lines, playing with capitalization for either the first word or all words in the name, and stripping punctuation from the selected nickname. Eventually, after endless updates to the boring-word-blacklist, I limited the nickname to words longer than three characters. I also ended up adding the



Brontes and Keats to the list of poets for variety.

PAYOFF

「Existence」

The storm is fast descending, existence seems
a summer eve,

「Long」

Tried, so well and long, we cannot hear each
other speak.

「Wrens」

queen of the wrens -- keep dry their light from
tears;

I'm not sure what the moral of this story is but this was a fun afternoon of coding, and I still occasionally pop back to the generator to flip through results. As with all generative art, each result feels unique and transient, and I have an entire folder of screenshots attesting to the fleeting sensation of having bottled sunlight, when a particularly good name pops up.

Maybe the moral is, listen to Friends at the Table! It will make your life better in surprising and unexpected ways. Or maybe it's that the intersection of disparate influences creates deeply interesting art. Or maybe it's, hey, go read some Emily Dickinson.

Code: <https://github.com/shannonkao/excerpt>

Demo: <http://excerpt.surge.sh/>

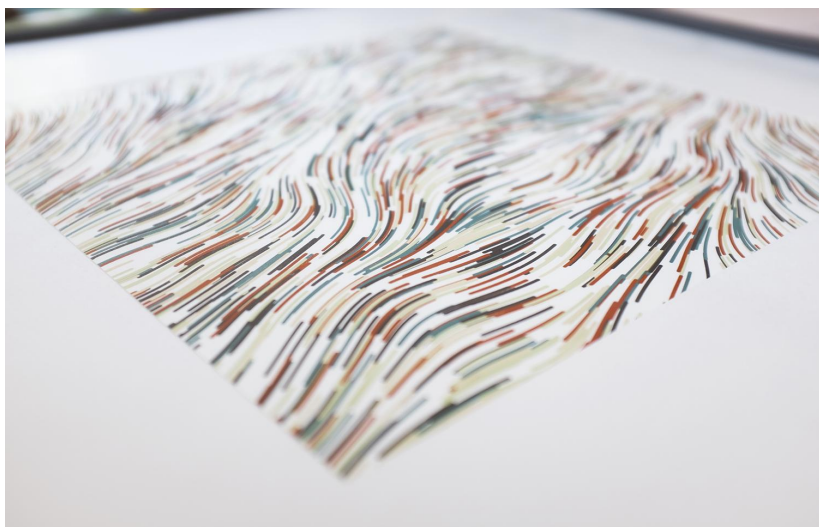
How GASP Gallery enables generative artists

By Neel Shivdasani

<http://gasp.gallery/>

A lot of generative artists seem to suffer from a sort of identity crisis. I can't even count the number of times that I've heard a generative artist say that they're "not an artist," or that they're just a coder. The fact of the matter is that generative artists can make incredible images that viewers wouldn't distinguish from other types of art.

I think this disconnect has two causes. First, artists are often their own worst critics, and it's easy for generative artists who don't feel like their work has value to just dismiss it as code instead of art. The second reason is that it can be difficult to turn generative art into the sort of high quality physical output that people often associate with fine art.



That's why we created GASP Gallery. It's a platform that enables generative artists to sell high quality unique prints of their work to customers through an engaging and fun customization process. Buyers come to our website, select a style that they like from one



of our artists, and then get a totally unique version of that style generated for them on the fly. The buyer then has the opportunity to customize the color, texture, and other characteristics of the art. They can also generate new versions, share the results with friends and family, or save them to finish later. Once they've customized the art to their liking, they can order a high quality print that gets shipped directly to their home.



Our hope is that we provide a service that is beneficial to both art buyers and to generative artists. Buyers can easily and cheaply buy unique art that is suited to their liking, and generative artists can reach a new audience without having to take on the overhead of printing, marketing, etc.

If this sounds like something that would make sense for you as an artist or buyer, please come check us out at <http://gasp.gallery>.

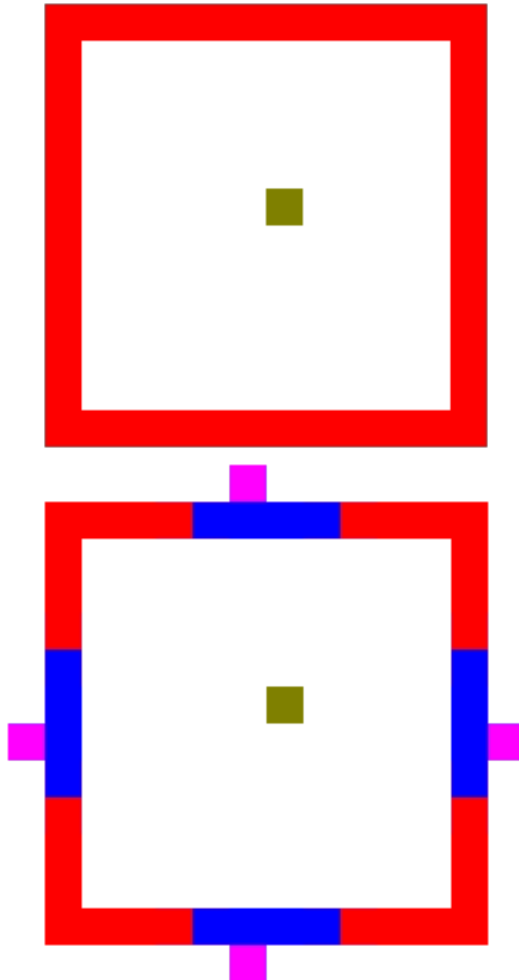



Unbounded Prefab Based Dungeon Generator for Tiled Game Worlds

By Justas Dabrila

<https://ssstormy.github.io/> | @justas_dabrila

Writing a generator that allows artists and level designers to easily tweak it is hard. I came across a viable solution while digging around Starbound that is not constrained by a generator-wide grid size and requires no custom tooling (sans the generator itself):





Have a pool of "prefab" dungeon parts that we know how to "place" into the world, then somehow "stich" an arbitrary amount of those parts together. This strategy consists of two algorithms:

The first is the stiching: how we chose which prefab to place and where to place it. Suppose we want to create a dungeon that is a linear corridor from start to finish. In order, to connect a prefab to another prefab, each prefab will need to have one or more doors (we'll get to defining doors in a moment).

The stich algorithm for this dungeon is best communicated in psuedo-code:

```
(1)
...

place_room :: (position : Position, room : Room);
pick_next_room :: (pool : RoomPool) -> Room;
find_suitable_door :: (in_room : Room, for_room : Room) -> boolean,
Position;

dfs_to_finish :: (room : Room, remaining_recursions : int) ->
boolean {
  while(true) {

    pool : RoomPool;

    if(remaining_recursions == 0) {
      pool := finish_pool;
    }
    else {
      pool := room_pool;
    }

    next_room := pick_next_room(pool);
```

```

        // we iterated over all the rooms!
        if(!next_room) {
            break;
        }

        did_find_door, door_position := find_suitable_door(room,
next_room);

        // couldn't find a suitable door, try another room.
        if(!did_find_door) {

            continue;
        }

        place_room(door_position, next_room);

        if(remaining_recursions == 0) {
            return true;
        }

        return dfs_to_finish(next_room, remaining_recursions - 1);
    }


    return false;
}

room := place_room(starting_position, starting_room_pool);
dfs_to_finish(room, 4);

...

```

This algorithm will place a spawn room, create 4 rooms and one final room that are each connected by doorways.



A subtle but very important detail is that `find_suitable_door_in` will find a random door in room at `door_position` and check if the `next_room` can be spawned there: is there enough space in the world and is any of that space obstructed?


In the case that we want to guarantee that a dungeon has a finish and we've failed to place the finish prefab, we can simply pick a new seed and retry the generation process.

We can tweak this algorithm to create slightly different dungeons that use the same prefabs in two ways: first, we randomly can mirror the room prefabs in the X and, if the art & level design allows for it, the Y axis. Second, we can reuse rooms by using a different stitching algorithm to place them at the locations of the doors we did not use (previous room didn't connect to it && next room doesn't connect to it) to create side-paths, treasure rooms, secrets etc.

An exciting part of this is that we can create a hierarchy of stitching algorithms. A room prefab could be made abstract into a stitching algorithm that generates a room. We could use this to replace our finish room with a stitching algorithm that generates some sort of interesting challenge room out of its own pool of room fragments, which, if we are crazy enough, are too generated by a stitching algorithm.

The second algorithm is the one that will place a prefab into the world. A block of text could be used, mapping certain letters to game things. This method can be cumbersome as you can only manipulate the text in the limited ways your text editor allows us to.

The a viable strategy seems to be storing the prefab as bitmap image which allows us to use any image editing software we like. During placement we can map pixel colors to game things: color (255 0 0 255) represents a solid, and color (128 128 0 255) represents a foe as



illustrated in Fig 1.

In order to represent a "door" within our prefab, we could treat certain colors as "logical" tiles that don't map to any game thing. Instead they provide information to our stitching algorithm. Knowing that, in order to create a directed doorway for a prefab, we could treat the color (0 0 255 255) as a door tile and color (255 0 255 255) as a direction tile. Our prefab will now look like Fig 2. This is sufficient door information for (1).

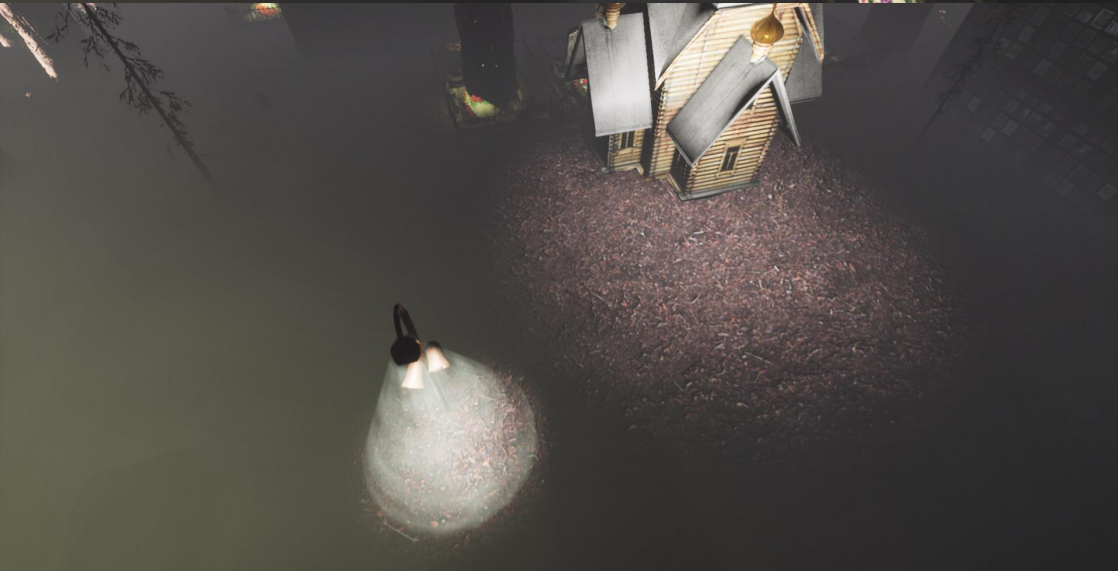
Infinite Graveyard Generator

By Mikhail Maksimov

<https://dyingfun.wordpress.com/>







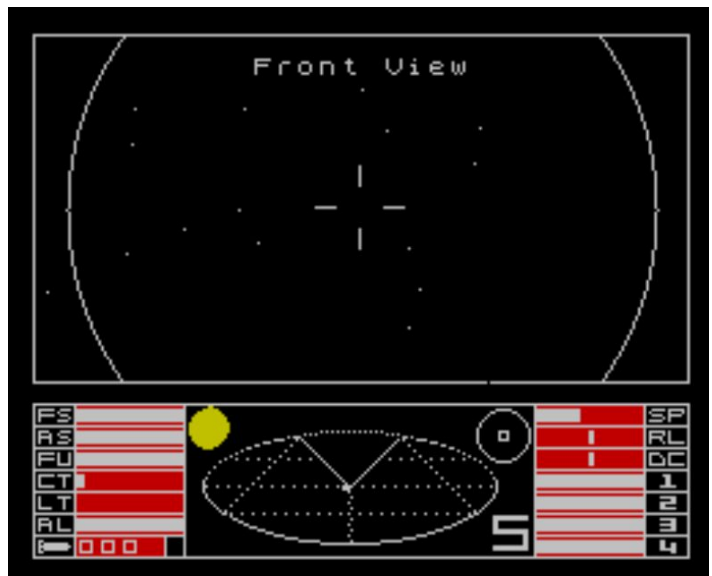
Procedural Real Scale Planets in Real-Time

By Alexander Zhuravlev

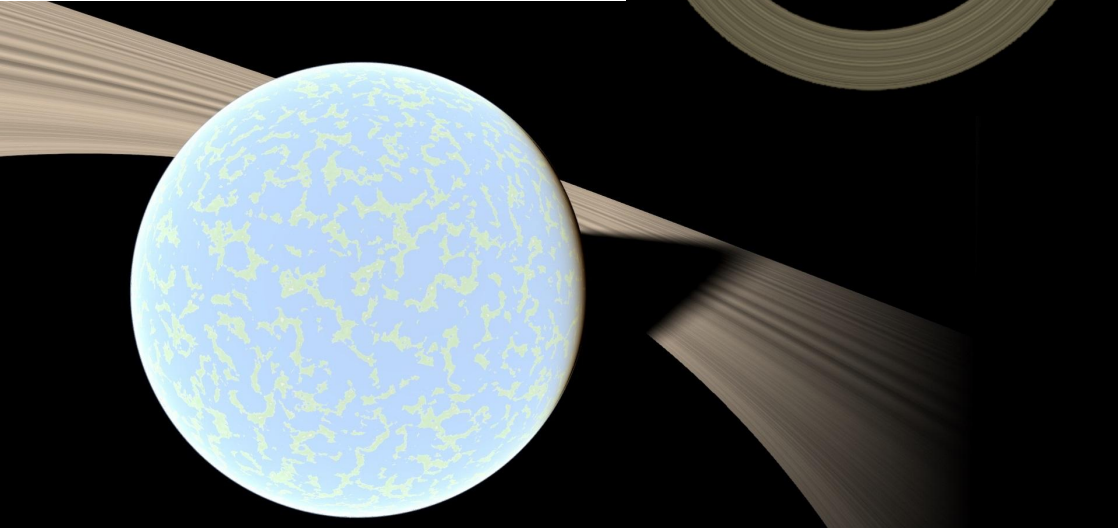
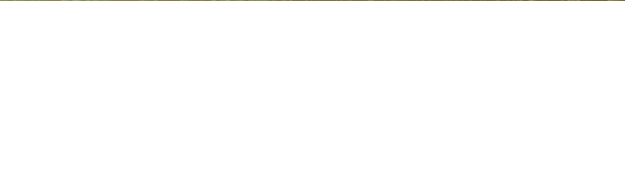
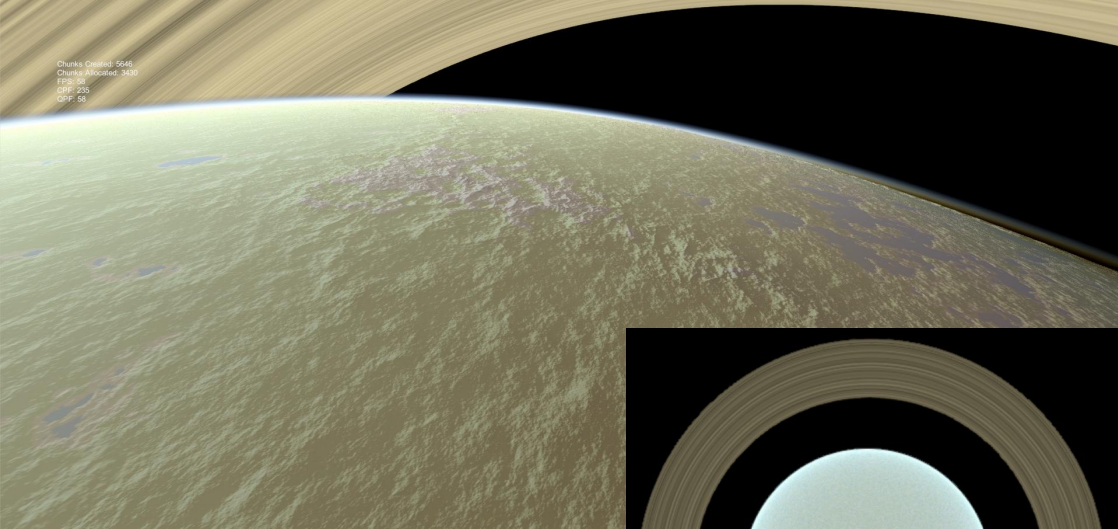
@alexander_tsuru

I've always been dreaming of creating a whole world of my own, a world where I can change or create something. And no limits at all: just your imagination and your creativity. Isn't it great to have a simulation you are able to adjust and customize any time you want?

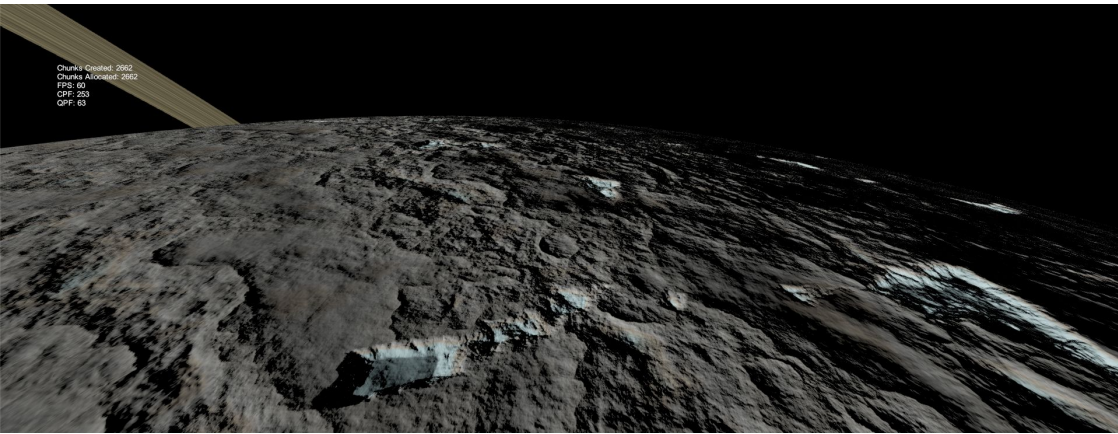
"Why not create something that we could only dream about just a few decades ago?"



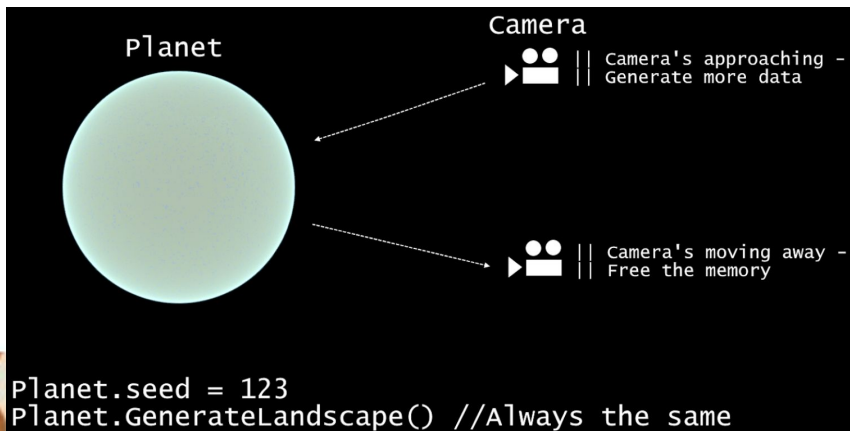
Well, I still remember the game called Elite 1984. It was a great concept: you could fly across the universe, trade, fight, explore. Back then, it was impossible to implement beautiful and detailed graphics. Planets were nothing but one-pixel circles. As time has passed, the computation power has increased sharply. We have very powerful CPUs and GPUs even at home. Why not create something that we could only dream about just a few decades ago? That's how I was inspired to create a procedural planet generation framework.



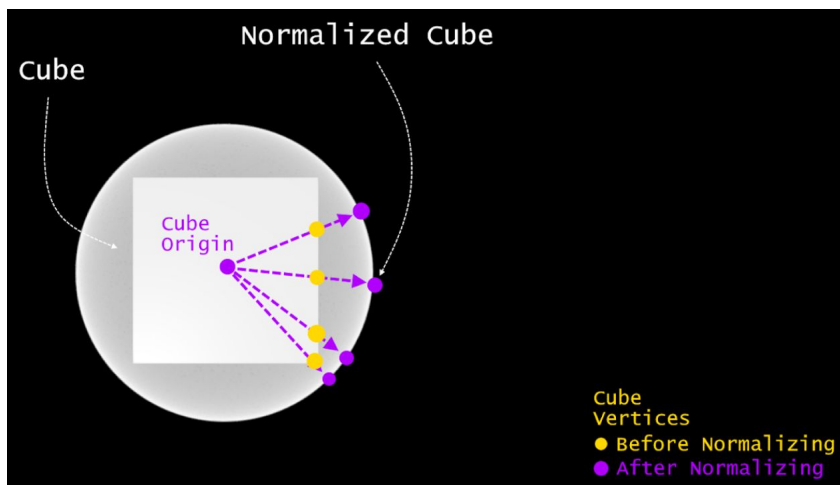
The whole idea behind this framework is to generate procedural, real scale and real-time planets. So, what does it mean?



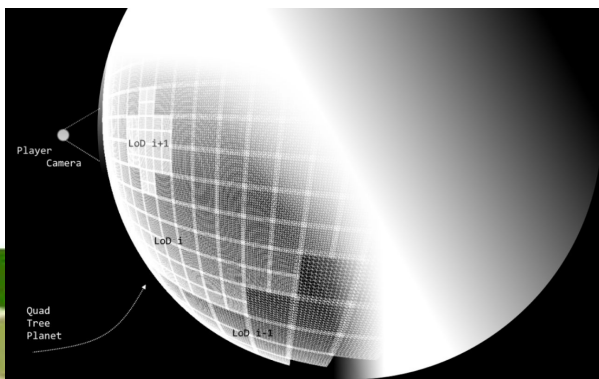
Worth mentioning, planets are not pre-generated, they are being generated procedurally in real-time with the camera approaching the surface. Planets are stored in RAM only and every time you visit the exact place it's always being generated from the seed value, keeping the same result. This idea is based on the pseudo-random functions which receive the seed value - state of the random. Just think about it: the whole game universe can be stored as a single number. Even assuming if it was possible to create the whole planet at once, it would require at least 300 GBs to store the planet on a hard drive.




Beginning creating a planet, the framework creates a normalized (each vertex vector is divided by its length) cube, resulting in a sphere. The cube faces are called chunks or quads and can be divided into four smaller chunks. All chunks have their LoD (Level of Detail) levels. To put it simply, the smaller the chunk, the higher its LoD level. We need this parameter to set up the chunk groups individually: for instance, it is very inefficient to create trees and grass when the camera is in space and the chunks' LoDs are low.

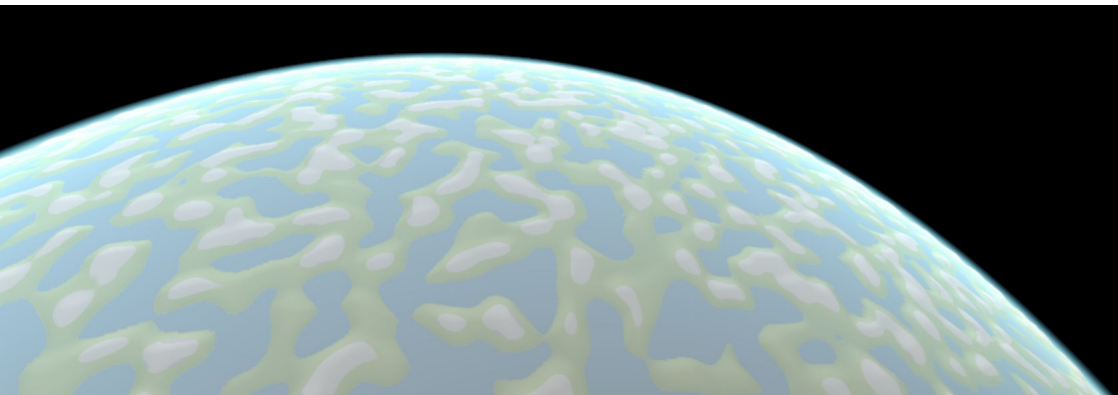


When it comes to a landscape, it becomes a little bit more complex. Firstly, let me explain the idea of Perlin Noise. Basically, that's a pseudo-random function that produces a wave-like graph taking the vertex position as the input. We can use the same function in 2D and 3D as well, generating a heightmap. Applying this map onto chunks we can get a simple landscape:

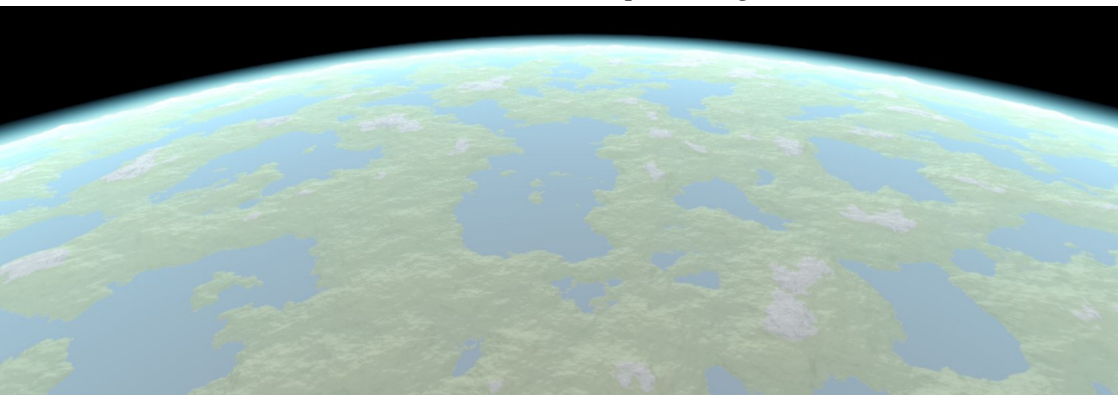





It becomes clear that the surface shouldn't be obviously 'repeating' in order to achieve something realistic or interesting to explore. That's why we use fBm (fractional Brownian motion) algorithm. Skipping the physics part, this algorithm creates the fractal noise (noise is being scaled and added a few times where each fractal is called an octave):



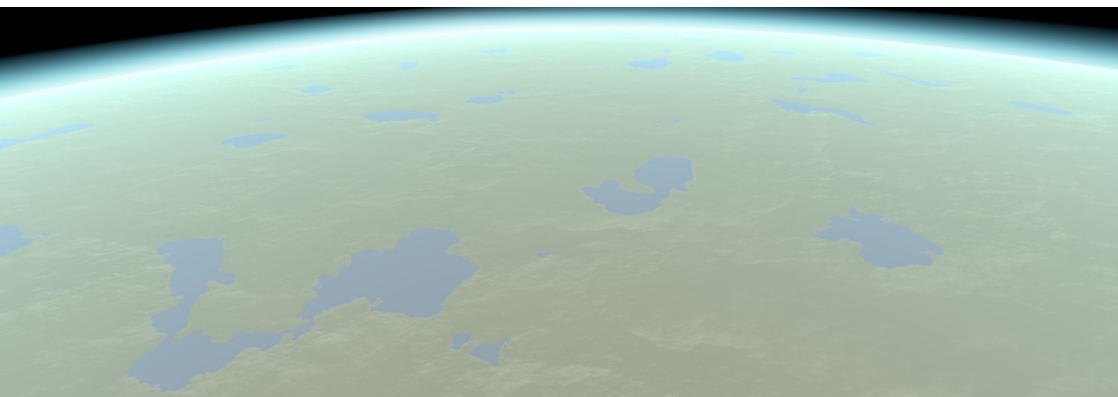
Now planets look better, but still, it's not enough. That's when other Perlin-based noise implementations can help us. There's a good number of them, but let's take only 2 examples: Billow and Ridged noises. Well, their names speak for themselves, the first one produces a 'billowy' noise and the second one creates ridges. Combining all these we can create something more complex and suitable even for the surface exploration games:





As for the planet rendering, we can create a height-based gradient with user-defined colors and color the planet. To render the mountains we also define an angular-based gradient. The atmosphere is rendered with a modified version of an accurate real-time technique described by Sean O'Neil in GPU Gems 2 (https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter16.html).

Additionally, you can create so-called 'addons' and modify the framework the way you want. As an example, the planetary rings are created by Planet Rings addon - a simple Unity component added to the planet object which hooks required functions from the generation process.



Want to try the generator yourself? A compiled demo project with a 100 km radius planet is available on GitHub (<https://github.com/wolfniey/Procedural-Real-Scale-Planet-Generation-in-Real-Time>). The demo works similarly to a benchmark, calculating your average FPS (Frames per Second). After finishing the test you can move around the planet as you wish. Use W, A, S, D to move and mouse scroll to change the camera speed.

Narrative Deckbuilding for Ambient Storytelling

By Jesse M. Porch

<https://jimporch.com/>

This is a simple proof-of-concept for a system that emulates the well-known “deckbuilder” approach to game randomization, keeping the intuitive model but taking advantage of a few digital enhancements. The goal is to provide a system that is easy for players to understand the possibility space while also giving the designer significant flexibility.

At its core, the Narrative Deckbuilder is a way of encoding discrete events in the form of cards, and assembling those cards into collections intuitively named decks. At defined intervals, a player will draw a card from a relevant deck and this card will resolve, modifying the game state as appropriate. Strategic actions on the part of the player can modify the cards present in the deck such as removing undesired outcomes or adding beneficial ones.

So far this closely resembles the common system used in physical games like Dominion or Ascension, where a player’s actions on a given turn are limited by what cards are drawn, with strategic modification used to make the average hand more beneficial. However, as mentioned, by implementing the system digitally, we can easily interact with these cards in ways that would be difficult in real life.

The two primary digital enhancements are deck composition and deck filtering. Composition allows a temporary merging of two (or more) decks for a single draw, then separating the decks seamlessly afterward. Filtering, understandably, means a draw is made from the subset of a given deck, ignoring cards that aren’t relevant to the specific draw while keeping the overall mechanics the same. These two abilities can be chained and intermixed, allowing complex scenarios to be addressed with the same basic metaphor.

Example: Consider a strategy game where a player manages a city


“The goal is to provide a system that is easy for players to understand the possibility space while also giving the designer significant flexibility.”

over the course of several years. Each week, one or more random events can occur, which are represented by an “Events Deck.” Some of these events are relevant only in certain seasons, so the cards are tagged with such information, and every turn a draw is made that uses a filter so that it only can draw events that are tagged for the current season. Additionally, the player’s actions can directly influence which cards are present to be drawn—performing certain tasks around the village can cause specific cards to be added or removed from the deck. Spending a turn stockpiling food, for instance, might remove one of the “Winter: Famine” cards from the deck, averting a future crisis. Though it might also insert a “Bandit Raid” card if the village’s defenses are not sufficient to safeguard the new supplies.

Future Plans: The above represents the core ideas that make the system work, and I think a fairly complex event system could be implemented with the model described above. However, there are several other options worth pursuing that would open new mechanical options without adding too much complexity to the system. First of all, individual cards could track their own state, storing information and changing throughout play. This could allow a single card to have different effects when drawn sequentially, like having gear that decays with use and must eventually be repaired or replaced before failure. Secondly, adding event hooks for cards to trigger actions during resolution would open new possibilities, like cards that remove themselves from play rather than merely being discarded. This would also allow cards to directly impact world state, allowing them to be more than just story snippets.

Conclusion: Obviously the system described above is very limited, and it can’t technically do anything not handled by an appropriately crafted roll-on-a-table system that is aware enough to filter out invalid results. The benefit, in my mind, is more in terms of a model that is easy for the designer to build out while still powerful and flexible enough to be worth using, and it provides easy hooks for the

“The benefit, in my mind, is more in terms of a model that is easy for the designer to build out while still powerful and flexible enough to be worth using...”



players to understand the working of the systems by portraying the random mechanics in a form they intuitively understand. By defining actions via the “merge and filter” model, new elements can easily be added into an existing system and either kept separate or integrated with others just by adjusting the specific card-drawing commands.

A barebones implementation of this model is available here, which I hope to be developing in the coming months to implement some of the enhancements described above.

You can find the Git Repo and take a look at the project through this link: <https://gitlab.com/porchjm/narrative-deckbuilder>

The Divine, in Verse

By Hodge

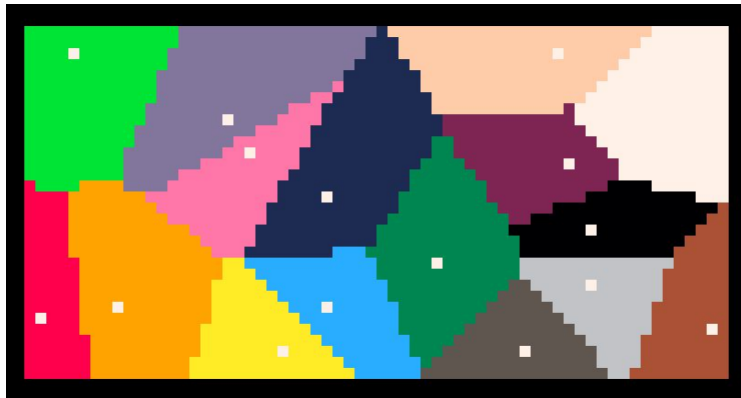
<https://harderyoufools.itch.io/> | @MrHodgepants

This year the University of Copenhagen and NYU ran a summer school on Procedural Generation. My project was a small Voronoi terrain generator in PICO-8. This article briefly skims over the major points due to size constraints, but

you can download the complete source code and play with an online version at <http://harderyoufools.itch.io/voronoi-park>.

WHY PICO-8?


While not free, PICO-8 is relatively affordable at around US\$15 and built-in mapping functions make Voronoi diagrams easy to implement. Also, it's a platform I know which is often helpful!



VORO-WHAT-O-NOI?

Wikipedia (https://en.wikipedia.org/wiki/Voronoi_diagram) describes a Voronoi diagram as "partitioning of a plane into regions based on distance to points." The Wiki page has more details on the formal definition but that summary is actually all we need. We'll implement it in two steps:

- 1) Generate a set of random points on our map; then
- 2) For each x/y co-ordinate on the map, figure out which of those random points is closest to it and mark it accordingly.



And that's it! When we're done we'll have a map divided into regions, each one containing the area closest to one of the points.

GENERATING THE POINTS

We'll store our initial points as Lua tables - specifically, their x position, y position, terrain type (which we'll get to later) and their point number (to keep track of them later on).

We're using a 128x64 sized map for our park, so the x/y coordinates are randomized within those ranges. For now, every point's terrain type is set to 'grass'. Here's the code to generate a single point.

```
function add_point(point_num)
    this_point={}
    this_point.x=floor(rnd(128))
    this_point.y=floor(rnd(64))
    this_point.typ="grass"
    this_point.num=point_num

    add(points,this_point)
end
```

To generate the entire set we wrap the previous function in a FOR loop:

```
function gen_points()
    for i=1,num_points do
        add_point(i)
    end
end
```

GOING THE DISTANCE

The next thing we need is a way to measure how far away our points are. There's two methods we can use. The first is Pythagoras' theorem, using the famous $a^2 = b^2 + c^2$ equation to calculate the

distance. The second is to use 'Manhattan distance' - a simpler, less accurate method which is basically Pythagoras' theorem with the squares removed. Manhattan distance allows for quicker generation but generates less natural, 'boxier' looking regions (this may be what you want!).



```
--pythagorean distance
function dist(x1,y1,x2,y2)
    return
    sqrt((abs(x1-x2)*abs(x1-x2)+abs(y1-y2)*abs(y1-y2)))
end

--manhattan distance
function mdist(x1,y1,x2,y2)
    return (abs(x1-x2)+abs(y1-y2))
end
```

ASSIGNING TERRAIN

We're using three types of terrain - grass, water and flowers. We need to tell the generator how much of each we need:

```
num_points=16  -- the number of points/regions in our
diagram
```

```
grass=8 --the number of regions to set as grass
flowers=3 --the number of regions to set as flowers
water=5 --the number of regions to set as water
```

Note the amounts for grass, flowers and water should add up to the total number of points, or Bad Things(tm) will happen. The demo cart has a basic UI limiting the numbers to sensible values, so it's best use that to change the numbers around. Once we know how much of each terrain we need, it's easy to assign the terrain to our points. Remember how every point was set to 'grass' when we generated them? That means we only need to change some of the points to flowers and water, according to the amounts we set above.

The code below goes through every x/y coordinate on the map. It measures that coordinate's distance to each of our original points; each time it finds a point closer than all the previous ones it sets the terrain value accordingly. This means an x/y coordinate's terrain may change more than once as it iterates through the points, but at the end it always has the terrain of the closest point.

```
function assign_terrain()
  for i=1,water do
    points[i].typ="water"
  end
  for i=water+1,water+flowers do
    points[i].typ="flowers"
  end
end
```


GENERATING THE MAP

This is it! We'll use PICO-8's `mset()` function to set the map values to to the terrain we set up earlier.

Once this is done for every x/y coordinate, our diagram is finished!

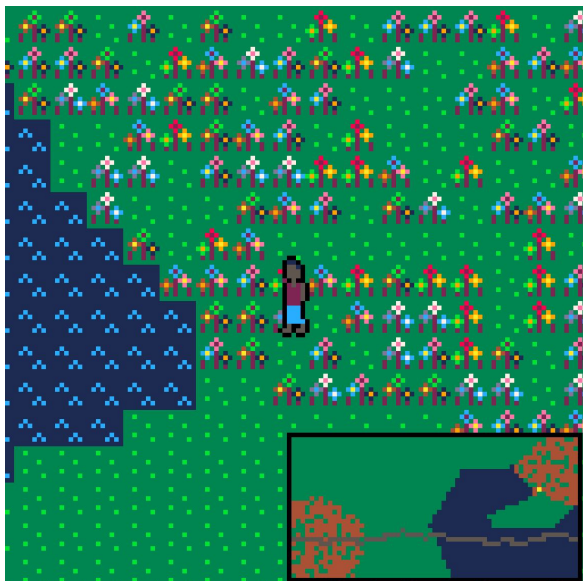
```
function set_map_values()
  --set map values
  for x=0,127 do
    for y=0,63 do
      local ldist=0
      local shortest=9999
      for i=1,num_points do
        ldist=dist(x,y,points[i].x,points[i].y)

        if ldist<shortest then
          shortest=ldist
          --set the map here
          if points[i].typ=="grass" then
            mset(x,y,1)
          elseif points[i].typ=="water" then
            mset(x,y,2)
          elseif points[i].typ=="flowers" then
            local ftype=floor(rnd(7))
            if ftype>4 then
              mset(x,y,1)
            else
              mset(x,y,4+ftype)
            end
          end
        end
      end
    end
  end
end
end
end
end
```

SHOWING THE MAP

This function draws a pixel map of our generated diagram. You'll notice it doesn't reference any of our code from earlier - at this point the diagram is stored in PICO-8's map data and we just need to read it with the `mget()` function.

```
function draw_minimap(mmx,mmy)
  for x=0,63 do
    for y=0,32 do
      local tile=mget(x*2,y*2)
      if tile==1 then
        pset(mmx+x,mmy+y,3)
      elseif tile==2 then
        pset(mmx+x,mmy+y,1)
      elseif tile==3 then
        pset(mmx+x,mmy+y,5)
      else
        pset(mmx+x,mmy+y,4)
      end
    end
  end
end
```



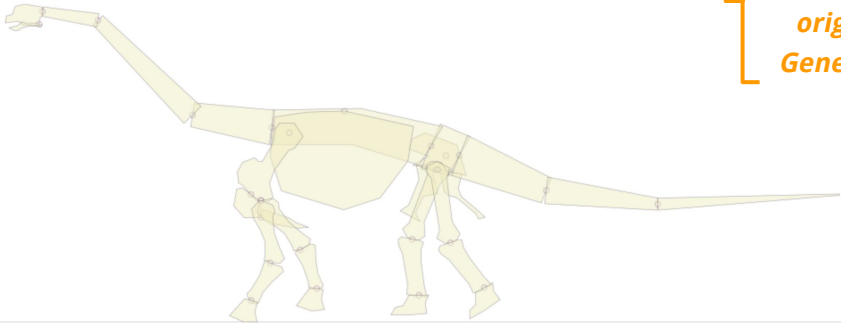
Writing programming languages for procedural generation

By Elle Sullivan

<http://thecreativeperiod.com/elle> | @THISISDINOSAUR

Sometimes I like to write my own programming languages. Sometimes I also like to do some procedural creature generation. And sometimes I like to write my own programming languages in order to do some procedural generation, using a technique called Language Oriented Programming.

So, what is language oriented programming? It's pretty straightforward, you first design and write a programming language that is tailored to the problem you want to solve, and then you write a program in this language to solve it. The general idea is that you spend as little time as possible on that final program, and most of your time on the language. The language should be so well suited to the problem that the program you eventually write in it is almost trivial.



*A Dinosaur
from the
original
Generator*

Example time. I'm particularly interested in procedural creature generation. For one ProcJam, I started writing a procedural dinosaur generator, focusing mainly on skeletons. I made some okay progress, but I found it tiresome. I was writing so much repeated code and syntax that had no relation to the dinosaur bones. So I came up with a solution: to write my own programming language.


*A small snippet
of an Anatomy
program,
defining a
dinosaur*

This language, called Anatomy, lets me concisely define bones and how they can change in relation to one another, letting me define a dinosaur skeleton that's capable of morphing into any dinosaur. But, it let me do it in a way that's readable even to non-programmers, and, when I'm done, I won't just have a dinosaur generator, but also a tool that could even be used by non-programmers to make similar generators, like a primate generator, or a fish generator. It could even turn into a general animal or creature generator. People might in theory find ways to use it to make things outside of creatures that I never envisioned.

```
44 coracoid = [-10.21, 1.36, 0], [-67.321, 3.44, 0], [-93.161, 36.04, 0], [-59.164,  
45 scapula~coracoid = [23.8, 68.0, 0] ~ average(0,1), -90  
46  
47 sternalPlate = [-2.72, 5.44, 0], [-73.441, 2.72, 0], [-95.2, 8.16, 0], [-128.52,  
48 coracoid~sternalPlate = 2 ~ last, 15  
49 //body end  
50  
51 //neck start  
52 cervicalLastEndHeight = distanceBetween(dorsalSpine[0], dorsalSpine[last])  
53 cervicalFirstStartHeight = 87  
54 neckHeightDifference = cervicalLastEndHeight - cervicalFirstStartHeight  
--
```

In many ways, it's really not that different to more traditional programming, gradually building up functions, classes, and other abstractions to slowly build a more complicated program from different building blocks. I think it can really help manage that growing complexity. By explicitly defining the interface of your language, it means you maintain a separation between the machinery of generation (i.e. how things are generated) and what you are actually generating. It means the actual bit that's interesting, the bit that says what is being generated, is clean and simple, and accessible to as many people as possible.


This is just one way to use custom programming languages for procedural generation, you can take the concept even further. One project I'd like to attempt is an egg generator. Eggs can mostly be



described pretty simply, e.g. “A large spherical white egg with blue speckles”, leading to the idea of an “egg description language”. So the user would write something like the description above, and that would be the program, which, when run, would produce a nice image of an egg.

Make add on items more visible Make app start up times longer


But of course, I want the computer to choose what kind of egg to generate. So I could write a program that generates these egg description programs, and have the language just be an intermediary, never to be written by humans. But how are you going to write this program? I in fact have a custom programming language to write programs that output arbitrary descriptions, called ‘Terms’. I originally wrote it for a hackday at work, to generate ideas for experiments. It’s pretty simple: a Terms program consists of a file defining the structure of the description (e.g. ‘a [size] [shape] [colour] egg with [spots]’), and then a file for each descriptor listing the possibilities (e.g. for colour “white, blue, green, ecru...”). When run, it then selects a possibility for each term, producing a description as output (in this case a description of an egg). This can then be used to trivially produce output that is a valid egg description language program, which can then be run to produce an actual egg.



*Example
output of
my Terms
program to
generate
experiment
ideas*

So, by applying language oriented programming to an extreme degree, I can end up writing two different custom programming languages just to make our egg generator, but in a way that actually makes sense and still saves time.


Another nice benefit is that the intermediary step, the descriptions of the eggs, is useful itself. You end up with not just an egg, but an egg you know meaningful information about. When applied to other



projects, this means you can potentially end up with lots of useful information that other proc gen approaches wouldn't have given. E.g. it should be quite easy to write my dinosaur generator so that I know meaningful information about what the outputted dinosaur was like, such as what dinosaurs it's related to, where it might have lived, what it might have eaten, etc.

#lang terms

[verb] [noun] on [location]


*The
structure of
the
experiment
idea
generator*

Paper Terrain Generation

By Neil Bickford

www.neilbickford.com | [@neilbickford](https://twitter.com/neilbickford)



Here's one of my favorite unusual methods for generating digital terrain, using a piece of paper, a camera, and a computer.

1. Take a piece of paper - a napkin or a paper towel seems to work best.
2. Holding two opposite corners, crumple the piece of paper by pushing the corners together and introducing folds until you get a ball about five centimeters in diameter.
3. Gently unfold and spread the piece of paper. It should now roughly resemble a mountain range.



I like this method because it feels like the process of folding the paper almost simulates normal and reverse faults between tectonic plates, even though many things differ between the two.

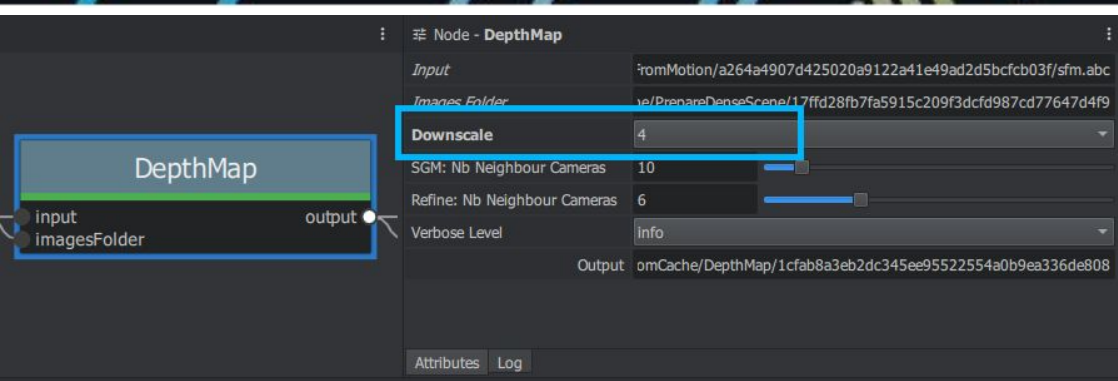
Make any adjustments to your terrain. Consider its apparent geological features, and how a larger terrain resembling this small-scale terrain could have been formed. Also consider how the terrain has been shaped over time, such as where people live and how they have changed it, if they exist.. Finally, adjust the boundary of the terrain, if it needs to lie flat or be lower relative to the highest peaks, for instance. This process is mostly symmetric, so you can also flip the piece of paper over to get another terrain.

We'll use photogrammetry to turn this into a digital terrain. I'll talk about using Alicevision Meshroom (<https://github.com/alicevision/meshroom/releases>) here, though other photogrammetry programs are also available. We'll only change two of the default settings here.

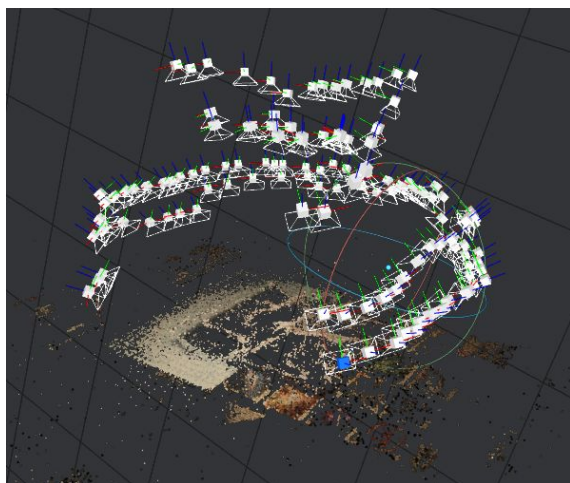
4. Take a series of photos of your terrain from different angles. I usually aim to take about 100-150 photos, and usually work my way around the terrain in a spiral from ground level to directly above the terrain. Try to keep the lighting and your camera's zoom level and exposure constant; having a textured surface underneath the terrain also helps reconstruction. You don't need a professional camera for this; a phone camera should work just as well.
5. Transfer your images to the computer.
6. In Meshroom, drag your images into the Images pane on the left-hand side of the screen.
7. In the Graph Editor on the bottom of the screen, remove the Texturing node at the right-hand side of the screen, since later steps don't require the texture of the paper.



8. Finally, select the DepthMap node in the Graph Editor and change the Downscale amount to 4 or higher, depending on the size of your images. This should speed up depth reconstruction.



9. Click the Start button at the top of the screen to reconstruct the mesh. This took from one to two hours for me.



Note: For the image at the start of this article, I also added a textured plane for water in the midground and background, some volumetric fog to show light rays and to convey distance, and vegetation using a hair particle system in Blender.

Once Meshroom finishes, you should see each of the reconstructed camera positions and a sparse point cloud on the right-hand side of the screen. Click on the MeshFiltering node in the Graph Editor and select the text in the Output field; this is where your reconstructed landscape is located on disk. You can now import this into a 3D digital content creation tool such as Blender, and remove everything except the landscape you generated. Now you have a digital terrain!

Modeling Possibility Spaces in Graph as a Tool for Generating Detailed Worlds

By Kelson Smith and Brandon Campbell

@kal_sar - @BrandonJCampbel

About the authors

We are a pair of full-stack software engineers who moonlight as game developers. In our day jobs we build enterprise systems that handle Big Data (woo buzz words!), and we believe that some of our experiences can be applied to the field of game design.

Our Theory

A procedurally generated world often starts as a set of modular content buckets. The form that the world ultimately takes is a unique combination of the content drawn from these buckets. The more content modules, the greater the array of possibilities. But there's a problem -- a meaningful experience is seldom random. Of those possibilities, there are a few interesting stories, wonderful characters, and fascinating worlds to be explored. Unfortunately, the probability that we will find the meaningful stories amidst the overwhelming noise is very, very low.

"This is the same structure used in neural networks used for machine learning."

We have a novel solution to improve the odds. By modelling meaningful relationships in a graph database, we can inject context into the generation process and increase the chances that our modular buckets will combine into something compelling.

What is a graph database?

We love the power of graph-driven architecture. Graph databases store information as a cloud of connected bubbles, the same way you might when brainstorming on a whiteboard. This architecture gives them the flexibility of a NoSql database with the query power of a relational database. This is the same structure used in neural networks used for machine learning.

Graph databases instead treat relationships as first class citizens, storing both individual pieces of data (Nodes) and the relationships between them (Edges in graph speak) as records in the database.

This gives us several major benefits, including:

1. Queries that involve exploring complicated relationships become nearly trivial in a graph database
2. Data is stored in a way that it maintains its context
3. As more varied types of data are added, the power of the graph to surface emergent patterns grows (while a relational database would simply become painful to work in).

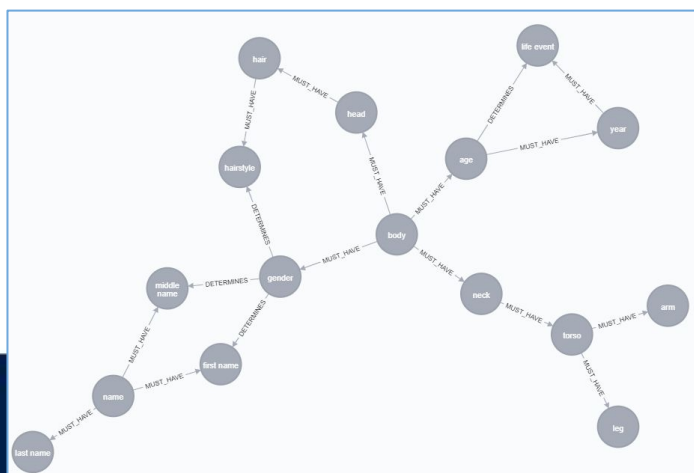
Graph is already seeing wide-spread adoption in many industries, from retail and medical to aerospace and social networks. Graph has an immense number of applications, but we believe that an underexplored area is it's potential in the creative field of games.

There are many graph databases out on the market, but we recommend Neo4j as an accessible entry point into graph development.

Modeling a possibility space in Graph

Graph databases are used in industry to synthesize large, disconnected data sets. This integration is governed by rules and patterns to sift out the meaningful relationships. We have discovered that we can reverse this workflow, modeling the basic rules that define a possibility space and procedurally generating an instance that follows this pattern.

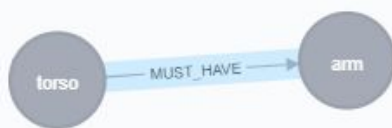
Here's a possibility space, describing the generation of a body:



With this approach we can model a hugely complex procedural system entirely in data. This allows us to tweak both individual parameters and large sections of our possibility space solely through the data, meaning we can model and procedurally generate any arbitrary subject matter using the same (relatively) simple algorithm.

Generating instances from a possibility space

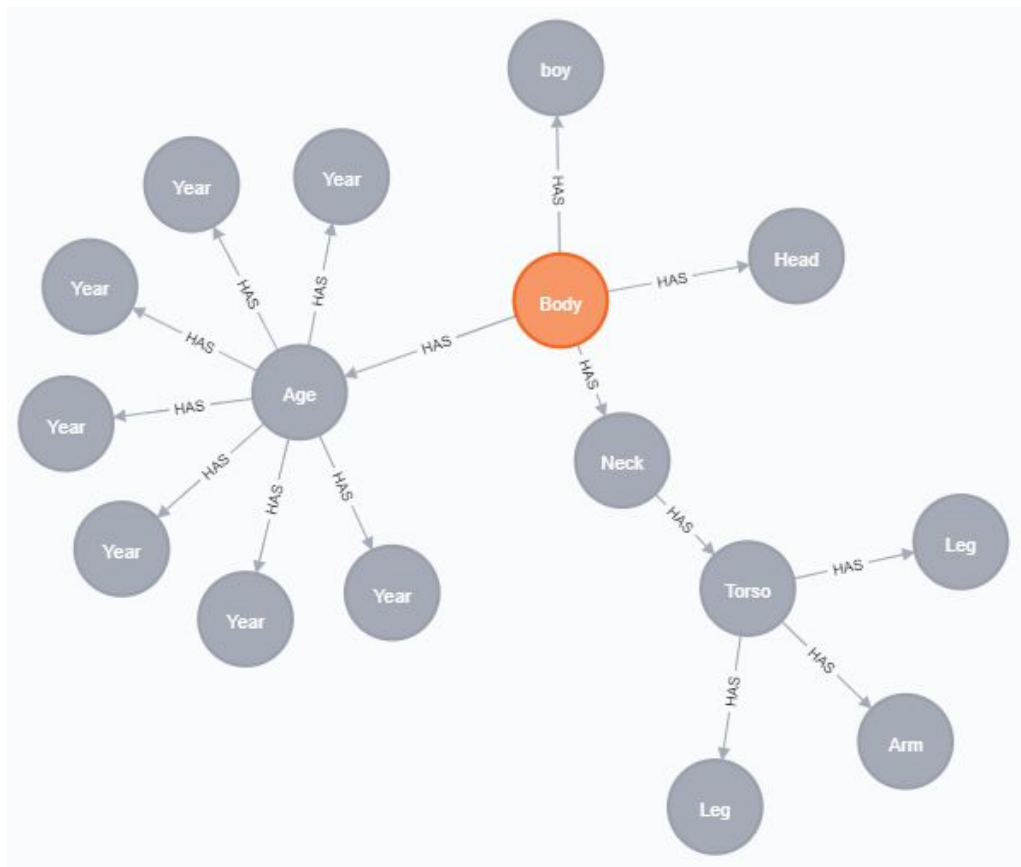
Think of your possibility space as a multidimensional cookie cutter and your instance as a multidimensional cookie. The dough is your content, sifted into buckets. The edges in the graph provide the shapes that your cookie could take. This strategy helps us to eliminate possibilities that may not make sense. Let's take a closer look at the torso and arm in our example space.



distribution: 1,2,100 **max:** 2 **min:** 0 **name:** torso **relationship:** MUST_HAVE **target:** arm


This model suggests that a torso might have no arms and as many as two, with a probability distributions shifted in favor of two. Most characters stamped out of this possibility will have two arms, but there is room for interesting outliers.

Consider this instance stamped our possibility space:



This is a unique character. He is seven years old, he has one head, two legs, and one arm. You can easily add personality traits and life experiences to the possibility space, allowing you to build deep characters whose unique history impact their AI.

This strategy lets you build your world in small, modular, but interconnected bits with an arbitrary level of granularity and complexity.



Making the game

With a freshly minted world stamped out of your possibility space, it is up to you how to use it. Your instance can be something like room layout and abilities in a roguelite metroidvania, story beats with a branching quest chain in a procgened open world game, or characters and motivations in a murder mystery.

While you can easily take your instance and use it simply as a starting point for your game state, keeping your game state in the graph can lead to some very interesting design options, as you don't lose any of the context you have generated.

Generators That Read

By Max Kreminski and Isaac Karth

@maxkreminski - @proc_gen

Most discussions of procedural generation have focused on either the things that are generated or the process that generates them. Less attention has been spent on the methods that generators use to interpret their input. Many generators take complex input and interpret it, and the process they use to read that input is often just as interesting as how they output something.

Recent years have seen an increased interest in approaches to procedural content generation that interpret and meaningfully respond to complex forms of input, often forms of input that were not originally intended to be used as input to a generator. Challenges such as the Settlement Generation Challenge in Minecraft have encouraged the development of context-sensitive generators, capable of taking an arbitrary Minecraft map and generating a settlement that fits that particular context. Projects like WikiMystery have used existing corpuses of open data as a foundation for the generation of murder mystery scenarios.

Nevertheless, there is a tendency to talk about generation as though it is primarily a process of writing or ex nihilo creation of artifacts, sidelining the sophistication of the parts that focus on reading complex input.

At the same time, many people implicitly assume that it is both possible and desirable to produce an objectively correct and unambiguous interpretation. We think that extracting machine-usable meaning from complex input necessarily requires a creative act of interpretation: the complexity of the input ensures that it could always be read differently. In fact, the "incorrect" readings of the input might lead to a generator that produces surprising and useful outputs.

For example, many of the novel generators created for National

Novel Generation Month draw on the same source texts—frequently including *Alice in Wonderland*, *Moby-Dick*, and *The Odyssey*—yet produce very different outputs. One reason for this is because each generator takes a different approach to "reading" its input text.

CALL FOR PAPERS

[REDACTED] the [REDACTED]
[REDACTED] games, [REDACTED]
[REDACTED] and [REDACTED]
[REDACTED] experiences. [REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED] enable [REDACTED]
[REDACTED] the [REDACTED] research [REDACTED]

We propose the notion of "generativist readings." A generativist reading is an interpretation of a text consisting of a set of rules for generating artifacts similar to or based on the text. Much like a proceduralist reading of an interactive text focuses on deriving meaning from the rules or procedures within the text, a generativist reading of a text wants to know how to produce more texts like it. When a computer processes complex input and constructs a model of how it thinks that text could be written, we say that it is performing a generativist reading.

When we create generators to produce types of artifacts that were previously exclusively handmade, we essentially find ourselves manually conducting a generativist reading of a corpus of examples. For instance, if a human reader was to read *Moby-Dick* and handcraft a Tracery grammar that uses vocabulary and sentence structures drawn from the book to produce sentences that sound plausibly as though they could have been written by Melville, the

resulting grammar would constitute a generativist reading of the text. Twitter bot creators often do something similar to this, gradually sublimating the source text into a statistical model.

Manual generativist readings may even be an instrument of critique: In Umberto Eco's essay "Make Your Own Movie" he proposed plot generation algorithms in the style of various filmmakers as a way of parodying those filmmakers' styles.

As an illustration, compare two similar erasure poetry generators: The Deletionist and blackout. Both of these generators turn web pages into poetry by erasing most of a page's text. The processes these generators use to write their modifications are very similar. Therefore, the difference between them lies almost entirely in how each generator reads a page's text prior to modification.

The Deletionist interprets the entire webpage as a single unit. It decides which words to erase deterministically, such that running it repeatedly on the same webpage will produce the same result every time. It chooses which words to keep by following one of several possible patterns.

On the other hand, blackout reads each paragraph in isolation and makes no attempt to coordinate its reading of different paragraphs. It uses part-of-speech tagging and probabilistic fuzzy matching of valid sequences of parts of speech, recognizing simple declarative sentences that could be formed by omitting words and selecting a valid sentence.

The differences between these generators is primarily in how they read their input. For generators that have distinct "reading" and "writing" components, it is possible to alter or replace one part without changing the other component and still get interestingly different results.

[This is a summary of our PCG Workshop paper "Generators that Read", which can be read in full at <https://mkremins.github.io/publications/GeneratorsThatRead.pdf>.]

ti

re
re la

re
ti ti mi
ti

re

Incorrect Tracery

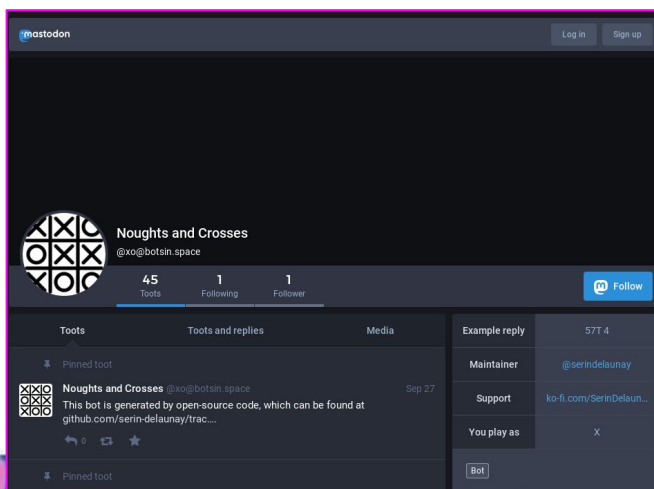
By Serin Delaunay

@SerinDelaunay

The official Tracery tutorial covers a lot of ground: word lists, madlibs, and recursion are powerful ways of generating text, and well-studied in the theory of context-free grammars. But the possibilities of Tracery and Cheap Bots Done Quick/Cheap Bots Toot Sweet are much wider than that! In this article I'll describe a few unconventional ways of using these technologies, with examples (mostly by me).

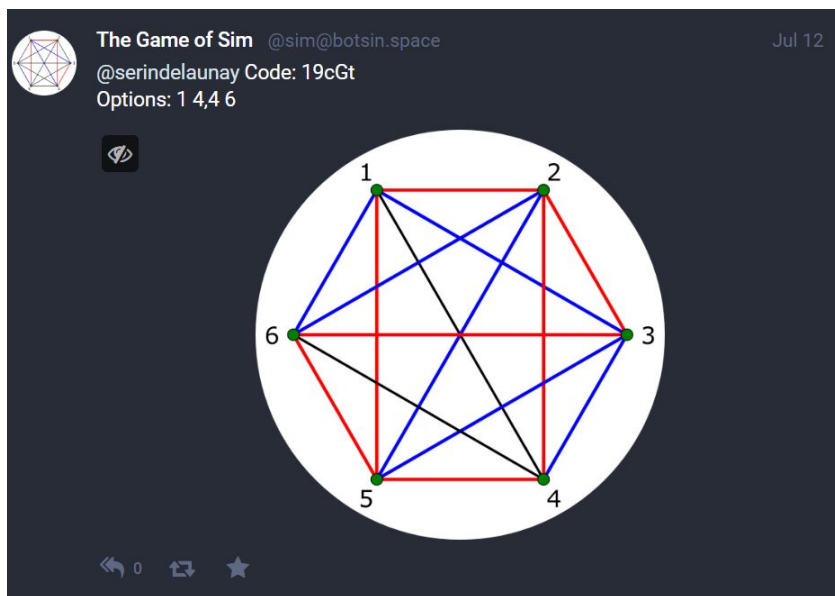
Large Generated Grammars

CBDQ/CBTS accept grammars and reply lists up to 4 megabytes. That's about 4 million characters, and easily enough for tens or hundreds of thousands of rules. It's enough to store an AI for the game of Noughts and Crosses [<https://botsin.space/@xo>], or the slightly larger game of Sim [<https://botsin.space/@sim>]. It's enough to store 17,000 fake piano scales [<https://twitter.com/DailyPianoScale>], or 11,000 snippets of news articles plus Tristan Tzara's method of generating Dadaist cut-up poetry [<https://twitter.com/DadaistTwit>]. It's enough to contain a few NaNoWriMo novels. Maybe you could even fit The Oregon Trail in it?



If you want to make a grammar that big, you'll need tools. Python is a great language for generating text, and has built-in support for the JSON format that Tracery uses. You could just run your bot entirely in Python, but with CBDQ/CBTS you won't have to worry about keeping your computer online.

If you don't fancy learning another programming language in order to generate code in Tracery, you can even use your text editor's find-and-replace tool to transform whatever raw data you have into Tracery format.



If you like Markov chain bots, you can use Cheap Markovs Traced Quick to easily make a Markov chain grammar that'll fit on CBDQ (depending on the size of your input text).

Saved Symbols

Saved symbols are described in the official Tracery tutorial. You can

use the syntax "[symbol:#other_symbol#]" to dynamically change the contents of the "#symbol#" tag in your grammar. You can do this repeatedly, and use "[symbol:POP]" to change back to the previous value. This has its limitations; a symbol call in this syntax will only be evaluated once, and the result will be saved. Nevertheless, the possibilities are endless:

- * Setting two or more symbols at once, allowing you to create groups of values. This is great for ideas like translations, inflected words, and opposites.

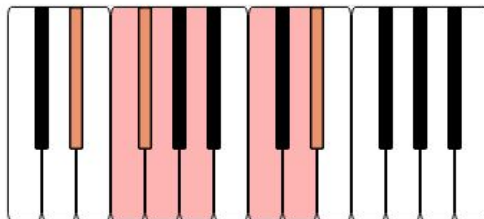
- * Swapping two symbols. If you set a "temporary" symbol to symbol X, you can set symbol X to symbol Y and symbol Y to the temporary symbol. This can be used for adding a little absurdity to a story, or extended to shuffling a deck of cards or generating Dada poems.

- * Hiding an active symbol. If you have a symbol that calls another symbol, or saves data to a symbol, you can turn it off by using "[symbol:]". Instead of changing the grammar, that symbol will now do nothing. You can turn it back on with "[symbol:POP]". This trick is the game-changer for Tracery programming.



Daily Piano Scales @DailyPianoScale · 7h

The scale of E flat recognised chromatic: E \flat F G \flat G A D $\flat\flat$ D E \flat . Play 4 octaves \downarrow then \uparrow , both hands, and QT with your fingering.

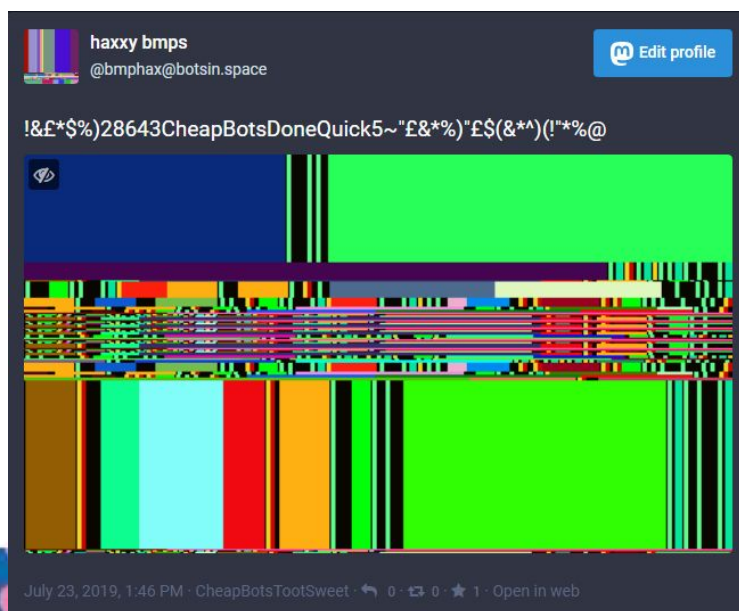


Switches

Turning an active symbol off and on is a step up from writing context-free grammars, but as a tool it's still blunt. A more flexible system is to have two symbols `switchA` and `switchB`, and only one of them active at a time. Then you can have a third symbol that calls them both: `"#switchA##switchB#"`. This switch could flip every time it's called, or be controlled by an independent symbol. This technique needs a lot of infrastructure if you want multiple switches to work with other, or if you want more states on your switches, but you've got the beginnings of a circuit board here and the sky's the limit!

Binary Data

CBDQ and CBTS have syntax for rendering images from SVG data. SVG is a text-based vector graphics format, and it's great for retrieving other images, stretching, filtering and juxtaposing them, and rendering text and polygons over them. Alternatively, you could use Tracery to generate a binary-format image (I recommend BMP), write it out as a percent-encoded data URI, and put it in an SVG `<image>` tag. [<https://botsin.space/@bmphax>]



It would be great to see files in other image, audio, and video formats generated or altered by Tracery. Their compression schemes and Tracery's limited methods of computation could combine into some strange sights and sounds. Hopefully CBDQ is able to upload databended GIF/video!

Javascript in SVG

Also, SVG supports embedded Javascript, so really you can draw anything you like without having to worry about trying to turn a grammar into a circuit board. Derek Ahmedzai's <https://botwiki.org/bot/trumptaxbot/> was a good example, using Javascript to make some simple calculations and show the result.

Turing Machine

In some versions of Tracery (unfortunately not the version that CBDQ's backend uses), a bug allows you to work around the limitation on active symbols when saving a value. So you can, for instance, pick two words from a list, string them together, and call the symbol with that name. Or repeatedly push symbols onto a pair of stacks and use them as an infinite tape. Or write a Turing machine transition table, and use my compiler [\[https://github.com/serin-delaunay/tmtracery\]](https://github.com/serin-delaunay/tmtracery) to make a Tracery grammar out of it.

Tracery is Turing-complete.

Example machine

```
{
  "function": "Accepts a string iff it is empty, or starts with 0 and consists of alternating 1s and 0s."
  "states": ["A", "B", "Y", "N"],
  "symbols": ["0", "1", "_"],
  "blank_symbol": "_",
  "start_state": "A",
  "accept_state": "Y",
  "reject_state": "N",
  "delta": [
    [ ["A", "0"], ["B", "0", ">"] ],
    [ ["A", "1"], ["N", "1", "_"] ],
    [ ["A", "_"], ["Y", "_", "_"] ],
    [ ["B", "0"], ["N", "0", "_"] ],
    [ ["B", "1"], ["A", "1", ">"] ],
    [ ["B", "_"], ["Y", "_", "_"] ]
  ]
}
```

Throwing Things

By Sean Butler

www.seanbutler.net | @butlersean

I DON'T LIKE PERLIN NOISE There, I feel better already!

This improved feeling comes at the risk of committing procedural blasphemy*. You'd be right in saying Perlin noise is very useful. Its also complicated and for some of us opaque.

A multilayered, random-ish waveform, Perlin and other** noise systems can be put to use making Waves, Islands, Caves etc. Noise systems generate these structures easily because they produce a repeating output that is never quite the same. Variations in the output are a similar scale and can have a distinctive shape, allowing us to use it as a source for hills or islands etc which are created by the same geological processes.

One way to improve the shape of the landscapes is to gradually increase scale on the y-axis, making the higher points of the landscape more pointy. A profile more closely matching mountains and hills.



Webmaster.vinarice CC BY-SA 4.0 via
Wikimedia Commons



Public Domain, via Wikimedia Commons

* An 'electric sinner' could use AI, and NLP to codify the rules of religious texts. Build a machine to break those rules perhaps automatically tweeting results. Maybe inciting others to commit sin and finally in action. Somewhat like a morality play (some might say that video games are already doing this better anyway).

High end procedural generation of sand dunes is complex with authentic outputs that may not appropriate for gameplay. Journey's gameplay relied on hand designed dunes. Extra rendering passes added interest. Conversely in Meteor Storm Escape we included a dune racing level, we compromised all authenticity for a challenging and exhilarating player experience.



We have a choice, from arbitrary algorithms and heuristics which generate forms useful for gameplay, to simulations whose internal factors accurately replicate the internal states and dynamics of the system found in reality whose aspects we find valuable for our game.

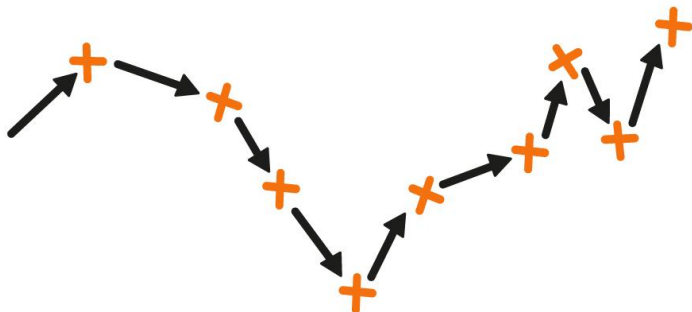
Plate tectonics allows volcanoes to form along long wiggly lines. Using a move-turn, loop to manipulate a sequence of vectors we can steer a random walk in a direction. Parameterising the length of the vectors and size of change in orientation, to create linear-ish path as the place where the world's crust is thin.

* Lets call all these value noise systems Perlin noise to avoid sentence mangling.

```

var rotation = constAngle * ((Math.random() + Math.random())-1)
Turn (0.0, 1.0, 0.0, rotation)
var distance = constDist * ( 0.5 * (Math.random() + Math.random()))
Move (0.0, 0.0, 1.0, distance)

```



This approach bears no relationship with the forces involved in the joining or separating of two tectonic plates, but can easily be tweaked to produce sequences which bear a resemblance to the jagged shape of the Mid-Atlantic Ridge or match specific gameplay requirements such as maximum distance between islands.

Assuming the direction and velocity of ejecta is random, but the angle of ejection follows a bell curve.

```

var angle = (( Math.random() * Math.PI/2) + ( Math.random() *
Math.PI/2)) /3
var direction = Math.random() * ( 2 * Math.PI)
var velocity = minvel + (Math.random() * 2)

```

Adjusting the range and distribution of these values has an impact on the shape of the islands produced. You might prefer more caldera, more sharp jagged peaks etc. Tweak and see what you get.

We could generate rigid bodies and run the engine's physics system. A simulation is not always an option on web or mobile. Instead

algebraically, one line of code calculates the approx landing point of that which is thrown out of the volcano.

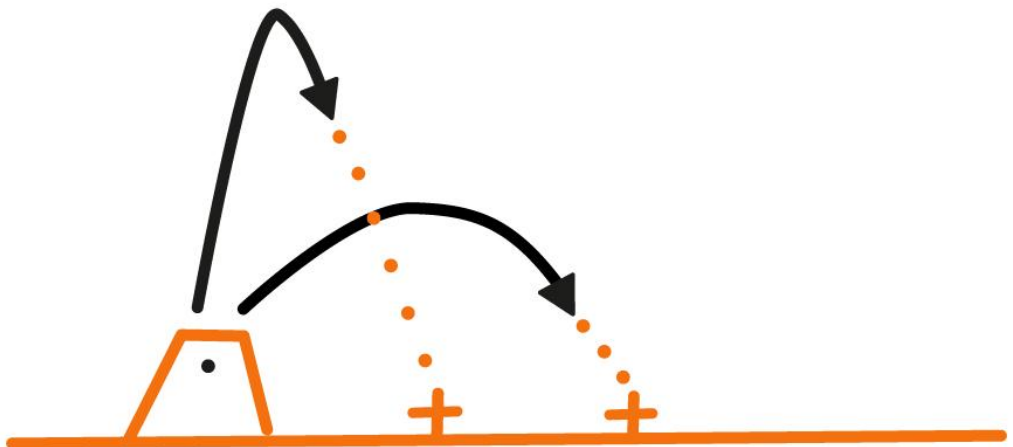
```
var distance = ( (velocity * velocity) * Math.sin( 2 * angle ) ) / 9.8
```

The distance away that something lands is proportional to the square of velocity times the sine of the angle it is ejected at, divided by the gravity. This assumes the ejection point isn't higher or lower than the landing point. Simple trigonometry will translate the distance and direction into a vector offset.

```
var deltax = Math.sin(direction) * distance;  
var deltaz = Math.cos(direction) * distance;
```

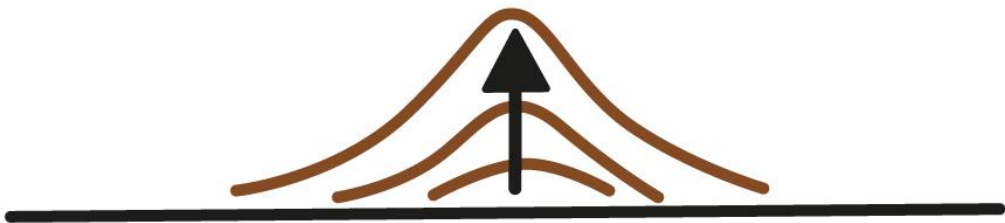
We can then do a little maths on to get a vertex index where it lands.

```
var coords = new THREE.Vector3();  
coords.x = Math.floor(centre.x + (deltax * (this.size * 0.5)));  
coords.z = Math.floor(centre.z + (deltaz * (this.size * 0.5)));  
  
var meshVertexIndex = ( coords.x * (this.size +1) ) + coords.z;
```

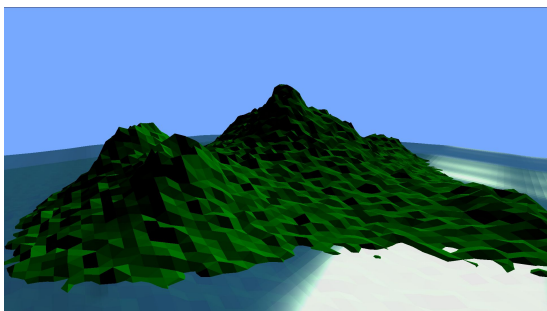
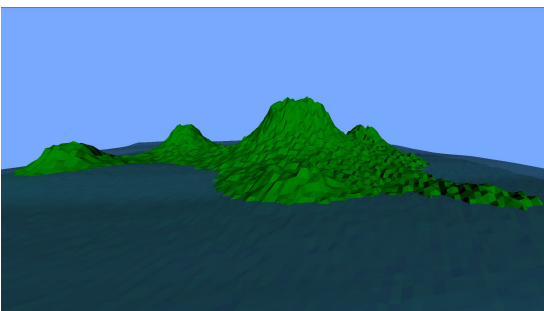
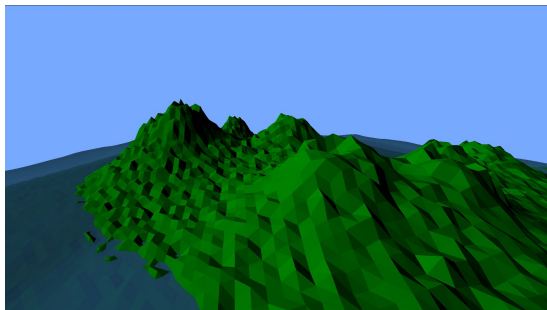
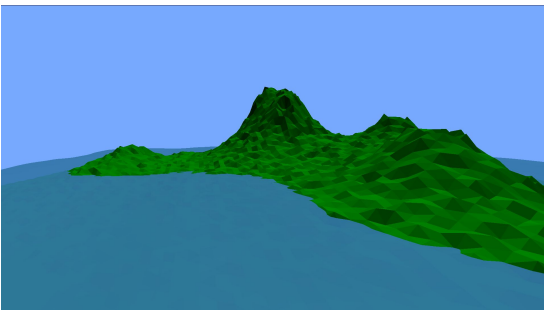


Finally raise the vertex a little!

```
meshVertex.y += 0.01;
```



The mountains generated have a various shapes, caldera, ridges, pointy etc. As islands they show convex and concave features around their edges.





ISS Earth Observations experiment and Image Science & Analysis Group, Johnson Space Center. Public domain via Wikimedia Commons

Producing accurate landscapes is complex and computationally expensive, noise systems difficult to tweak. Approximations are lightweight, flexible and easier to understand. The maths is simple trigonometry and random numbers. Simple algebraic simulations close the ‘Gulf of Execution’ associated with procedural generation.



Procedural characters for VR action game: 5 lessons learned

By Maria Mishurenko and Gordey Chernyy

www.bizarrebarber.com | @marmishurenko

When we started to design VR action game Bizarre Barber a year ago, we had a team of 2 people. We wanted to make accessible, fun action game that was inspired by our experience of moving to America and trying to grasp all the alienation and weirdness of immigration.

In Bizarre Barber, player play as the last human barber left after the apocalypse. People achieved singularity and can't afford to have bodies anymore. The wealthiest have heads, but hair absorb post-nuclear radiation and barbers are helping people] heal by cutting hair. We wanted to make the characters surreal and diverse - with the New York subway being the main inspiration. In fact, we set up all levels to look like bizarre mashup of several US subway systems.

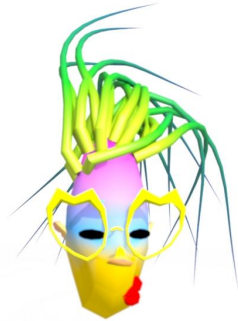
With all the gameplay / programming / VR testing problems, it quickly become apparent that we can't afford to individually model, texture and rig more than 200 character heads. So we decided to build character head / hairstyle generator in Houdini. When it was ready, we put the pipeline and characters to test and had some interesting discoveries.

Curation sometimes takes too much time

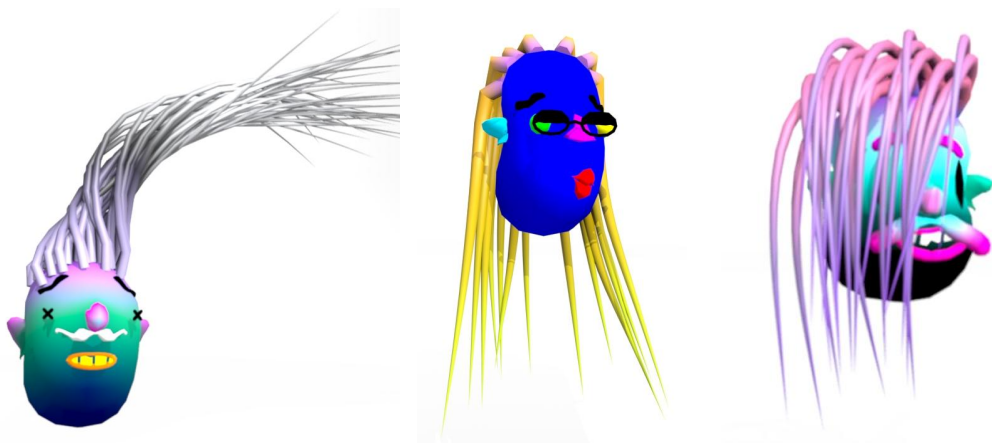
One of our reasons to build the generator was saving time. But, the process of sifting through dozens of generated heads to pick the best ones wasn't particularly quick. It was also surprisingly boring and non-inspirational for us as artists: it didn't allow any serendipity. So we decided to move on and include some of the manual adjustments, which led us to...

Trying more handcrafted approach to procedurality

We set up a "mother" character template and then through a series of Houdini takes iterated manually with settings. We rebuilt the



creation system the way it takes no more than 3-4 minutes to create an entirely different, manually adjusted character. That worked for a bit, but after creating 50-60 characters this way we realised that they tend to be suspiciously similar-looking!



Eliminating biases in art creation is important!

The easiest way to ensure true diversity without feature creeping the system was to invite as many different creators as possible. We invited the wonderful NYU Game Center community members and asked them to build characters for us. People start unconsciously building characters that looked like them and invent wonderful combinations of colors and shapes that we didn't think was possible! The problem was finally solved, or at least we thought so...

Playtesting will destroy (and also save) you

We set up a Houdini-Unity exporting pipeline (unfortunately Houdini engine doesn't work in real-time) and started to massively playtest the characters. Since the game have rhythmic nature, players always expect to find some patterns in gameplay design - that much we knew. But we absolutely did not expect to find that our idea of hundreds of completely different characters didn't resonate

with players! They wanted certain characters to repeat from time to time, to see them more than once, to come up with a story for them. It was very sad, but necessary to scrap lots of the good characters from the gameplay in favor of repeating the best ones. We also gave distinct voices to those who survived the art purge. That seem to improve the experience and made people care more about the characters. Eventually, we made more levels for the game than planned, so all the scrapped characters returned in other roles (as bosses and spectators).

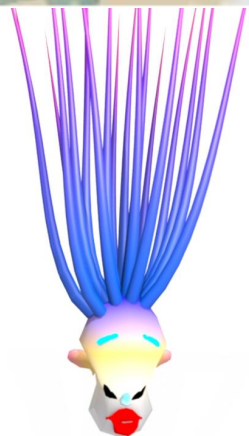
Final art direction pass is a glow-up for your procedural children

We had a lot of folks to help us making the game, one of our advisers was a prominent art director with lots of experience working in spatial mediums. He suggested to refine the color palette and decrease the number of traits / features to achieve sleek and distinguish look. For us, it was very important to hear, because the constant switching between technical design of a system and actual character creation was mentally exhausting. That was a constant battle of practical and creative mindsets and having an outside person to take a look and point at the shortcomings was very refreshing.

Using procedurality and creating custom systems for asset generation is very important for small indie team with a limited budget. However, there are tremendous risks and challenges to that, especially when designing tools for emerging media games, like VR and AR. Our main takeaway is the idea of developing the system alongside with regular playtesting. That will require to set up specific pipelines from the very beginning, but let you avoid extra work along the way.

Maria Mishurenko and Gordey Chernyy, Synesthetic Echo studio co-founders. Developers of Bizarre Barber.

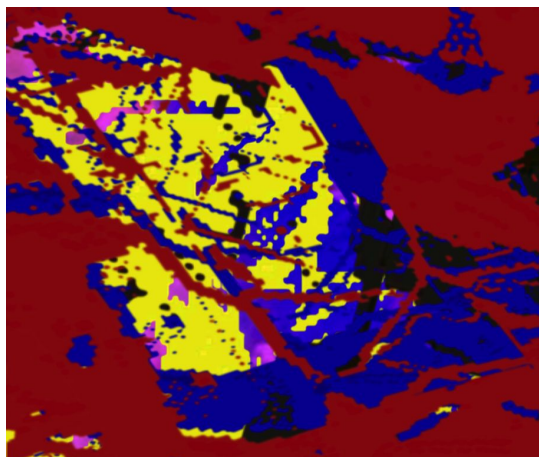
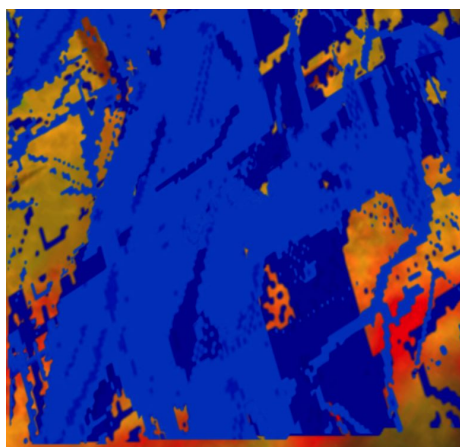
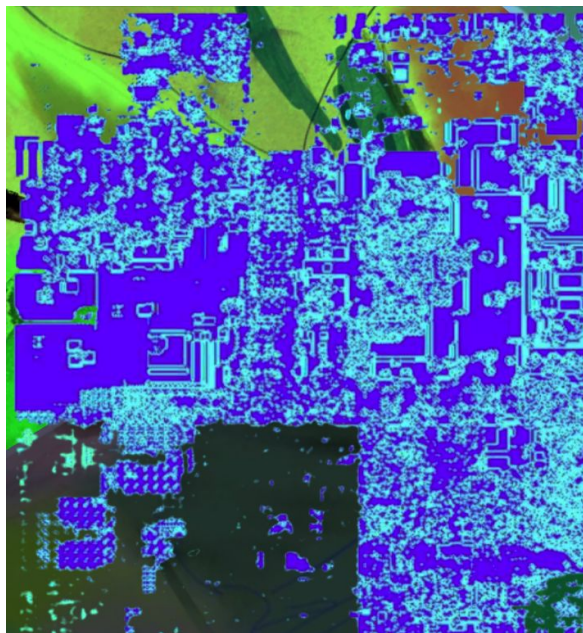
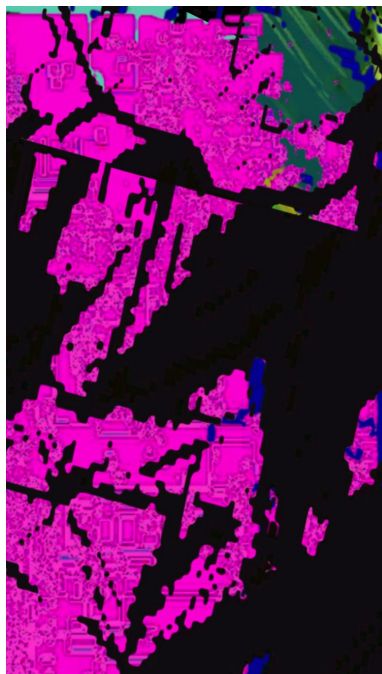
(images are renders from Houdini, with the approximation of hair look, as the hair are being automatically generated in Unity)

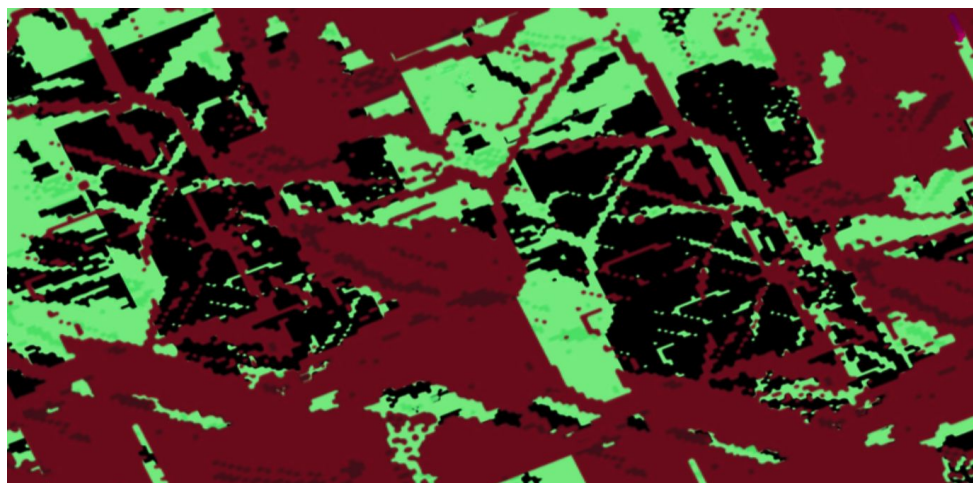
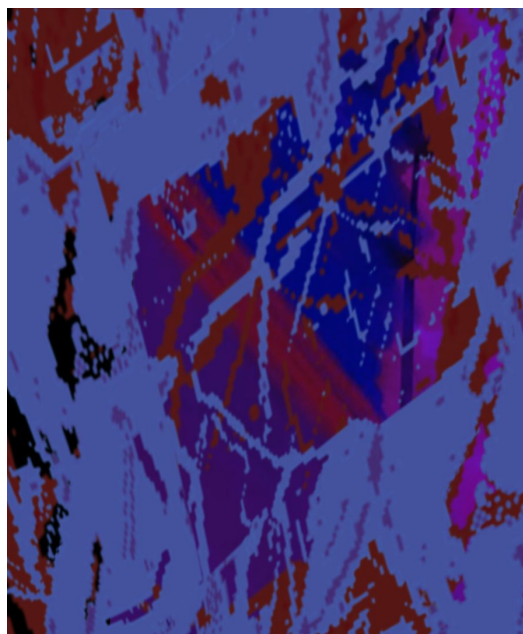
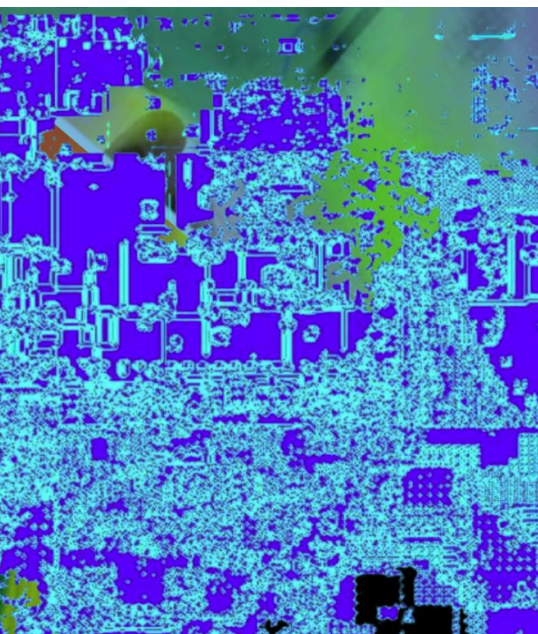


Paintmaster 5054 - Jazzfunk Greats

By Lee Tusman

<http://leetusman.com/>






Report on the First #InnopolisAI Art Contest

By Joseph Alexander Brown, Hamna Aslam, and Nikita Lozhnikov
@jb03hf, @Hamna72, @lozhn from Innopolis University @InnopolisU

The question as to if a computer can be said to be creative is a hotly debated question of philosophy. Some make appeals to the issues of the Chinese room in that computers can build things but not understand what they produce, making humans distinct creative forces. Others make claims based on the Turing test in which emulation should be seen as proving a property as esoteric and subjective as intelligence or creativity, and surely if the computer artwork cannot be told apart from a human creator, either they are both creative, or the concept of creativity has little meaning. As these debates rage, the news media has grasped on to single points and made definitive claims that computers are not of creative stock via an argument that their science editor happened to find in a press release from some University or another that makes one of the points and then states this is a definitive outcome. It implies that scientific debate is settled by public press decree.

As educators and creators of generative systems, we were confused by reports in our news stream which stated so definitively an answer to the question that is under debate in the Introduction to AI class at Innopolis University. It was rather heavy-handed to call the discussion to be closed. Hence, we asked our students to join in on this conversation and create their own works for an art contest to be evaluated by a set of humans, and the first #InnopolisAI Art contest was formed. For the second assignment for the Introductory Class on AI, we decided to run a contest with the following rules:

- The process implemented would need to be an Evolutionary Algorithm. This choice was made to make the content of the competition, similar to what was being taught in lecture.
- The process implemented should take a 512 by 512-pixel image as input. This was made to allow for a fitness function to be created. The size selected as several available test images were generally of this size. During the contest, a set of images would be



given as inputs, unknown to the developers.

- The process implemented should present a 512 by 512-pixel image as output. This was based upon the size of the input image.

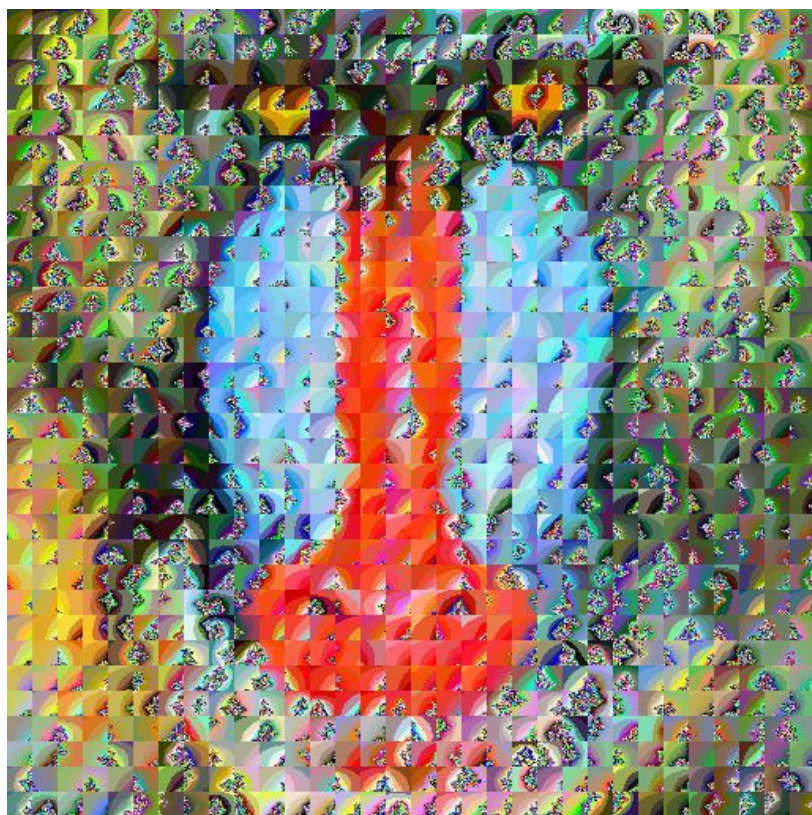
Other than these concerns, made primarily due to the available resources and the requirements of the course and curriculum design, no other conditions were set on the images created in the first stage of design. For the contest, we limited the set of base images for a generation. This limitation was to ensure a levelled playing field, a selected input image could inherently be more beautiful than that used by another student. Further, it would focus the design on generating an artistic creation without an assumed input image, meaning the created method would need to be general. The students would select one target image from the set and provide an output into an online collection system. They then were allowed to pick three images from their fellow students, those with the most selections would then be sent to a panel of guest judges. The guest judges were selected for their backgrounds in AI, PCG, Art, or a combination of the factors. The judges were presented with six images which had emerged as chosen favourites of the students and provided a ranked ballot. In the end, the scores were close. However, three finalists emerged that only scored a point away from each other.

Thank you to our guest judges: Daniel Ashlock, Mike Cook, Rhoda Ellis, Timur Fazullin, James Hughes, Elle Sullivan.

The class enjoyed the project-based method of creation and having the freedom in the implementation. The three finalists each received a high-quality framed print of their work, and the grand prize was a t-shirt with the winning image. The final announcement was recorded at the end of the semester and can be viewed at <https://youtu.be/ARSzT-EXuYM> and works in progress were posted to Twitter under the #InnopolisAIArt hashtag. The plan is to continue on this practice in the class, due to its educational value and the

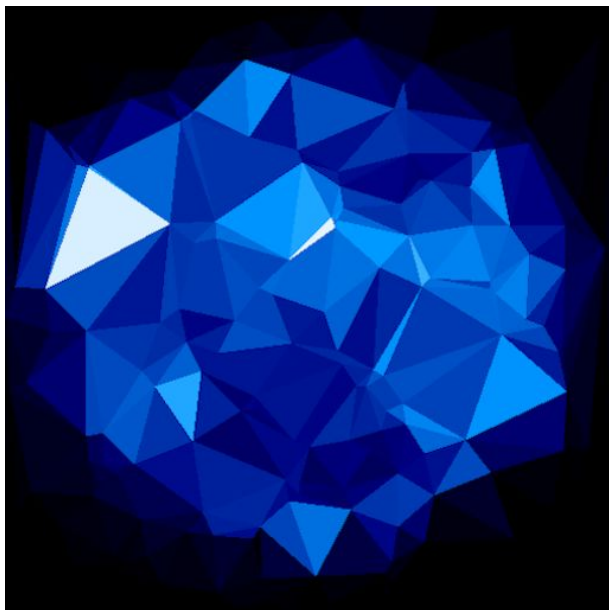
enjoyment in the learning process for all involved. We would invite anyone wanting to join the guest judges in future to contact the authors.

First place: Temur Kholmatov

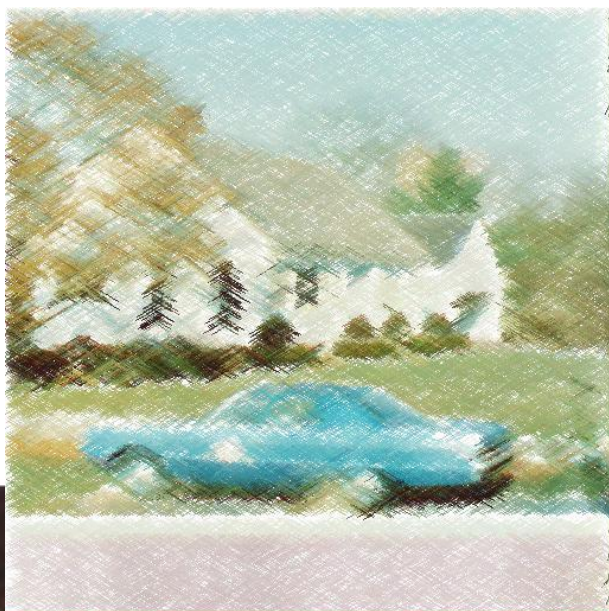




Second Place: Daria Miklashevskaya



Third Place: Kamil Saitov



Does Generative Poetry Need A Theory of Forgetting?

By Terry Trowbridge

<http://artbar.org/>

Does generative poetry need a theory of forgetting? Generative poets are familiar with substitution and erasure. Those are not the same as forgetting. The difference is hard to explain, for lack of a good definition of what it means “to forget.” For example, are repressed memories forgotten or erased? Are repressed memories just moved from a metaphorical desktop into a metaphorical partitioned drive? Are they forgotten if they can be recalled? Are they forgotten if they cannot be recalled but they still influence decisions? The question about poetry, though, can be partly answered by examining two different ways of representing subtraction: a poem that represents forgetting as a writing process, and a poem written specifically to represent erasing characters as a writing process.

This short theoretical essay is meant to pose a challenge. There are poems that were generated through an action of forgetting. There are poems generated through an action of erasure. Can we design a machine that generates poems through forgetting, rather than through erasure? In order to try, we need a working theory of forgetting that can identify a difference between erasure and forgetting in poems.

Erasure poetry has a burgeoning literature of primary sources and literary criticism, both for human generated and computer-generated poems. For the purposes here, whatever process of computer generation used, erasure means that the computer is a tool that substitutes an eraser on the previous generation of paper media. The computer system is a more complex tool for deleting text or, sometimes, can be seen as adding or subtracting space in a field that is like an enhanced page. Erasure poetry is something like an analog version of hypertext. Rather than use hypertext though, the goal is to contrast erasure with forgetting. Therefore, what slam poets call “page poetry” is useful, because poetry of forgetting is a human act on paper (so far), and therefore

the non-digital medium should be the starting point.

Gregory Betts' 2009 project *The Others Raised in Me: 150 Readings of Sonnet 150* is erasure poetry. Betts begins with the entire text, and spacing, of Shakespeare's Sonnet 150. He then generated 150 poems by erasing characters. The book is divided into 14 chapters, each chapter titled with the next line of Shakespeare's poem (the first chapter is titled, "O from what powre hast thou this powrefull might" etc.). Betts coordinates the loose meanings and forms of the poems to the chapter title.

Once the premise is explained, reading Betts' book for critical appreciation is synonymous with reading it to glean the procedure for each poem. The erasure is deliberate, probably. Where it is not deliberate, the poems are still sorted by chapter. For example:

85.

powre corrodes
by brute
hate (Betts, 2009, 133).

Each poem is unique. They are recognizable as poetry to a novice reader of Canadian literature, generally.

Poetry of forgetting is rare. There is an example in Eugenia Zuroski's 2019 chapbook *Hovering, Seen*, titled, "The Poem I Wrote in my head While Watching Dovzhenko's 'Earth' and Can't Remember" (Zuroski, 2019, 8). The text goes like:

a [something something], a [something
something], a [something], a
[something]

All of the poems in Zuroski's chapbook allude to an act of remembering. The title *Hovering, Seen* might allude to a quality of memory as a suspended image in the context of one's mind.

What makes the process of forgetting into the writing process, rather than an incidental challenge? Writers of any sort can sympathize with forgetting lines of text as a constraint on a writer. However, this poem is specifically written by forgetting lines of text.

First, the reader is alerted by the title. There are famous examples of poetic fragments that were cut short by forgetting, though, like Samuel Taylor Coleridge's *Kubla Khan*, which he supposedly could not complete because he dreamt the lines and forgot them in the process of waking up. Not so, with Zuroski.

For Zuroski, forgetting the lines is the content. In humans, forgetfulness is often accompanied by a sensation of a thing forgotten, once the poet realizes they used to know a line, or an image, and have lost it. The forgotten mental object is replaced by a feeling of its loss, and often, a sensation of its size or approximate number. Thus Zuroski can compare the set of [something something] to the set of [something]. The sensation of forgotten mental objects is produced on the page using square brackets []. They are not erasure, but a syntactical representation of the sensation of a forgotten fragment with the word "something" and the brackets representing the feeling or act of forgetting

The comparison of these two (or one hundred fifty-one), poems by Betts and Zuroski implies a difference between erasure and forgetting as procedures for poetic generation. Can that difference be examined and done by machine?

Works cited

- Betts, Gregory. (2009). *The Others Raised in Me: 150 Readings of Sonnet 150*. Toronto: Pedlar Press
- Zuroski, Eugenia. (2019). *Hovering, Seen*. Toronto: Anstruther Press.

