

# Polygonal Map Generation for Games



I wanted to generate interesting game maps that weren't constrained to be realistic, and I wanted to try some techniques I hadn't tried before. I usually make tile maps but instead used a different structure. What could I do with 1,000 polygons instead of 1,000,000 tiles? The distinct player-recognizable areas might be useful for gameplay: locations of towns, places to quest, territory to conquer or settle, landmarks, pathfinding waypoints, difficulty zones, etc. I generated maps with polygons, then rasterized them into tile maps that looked like this:

## Table of Contents

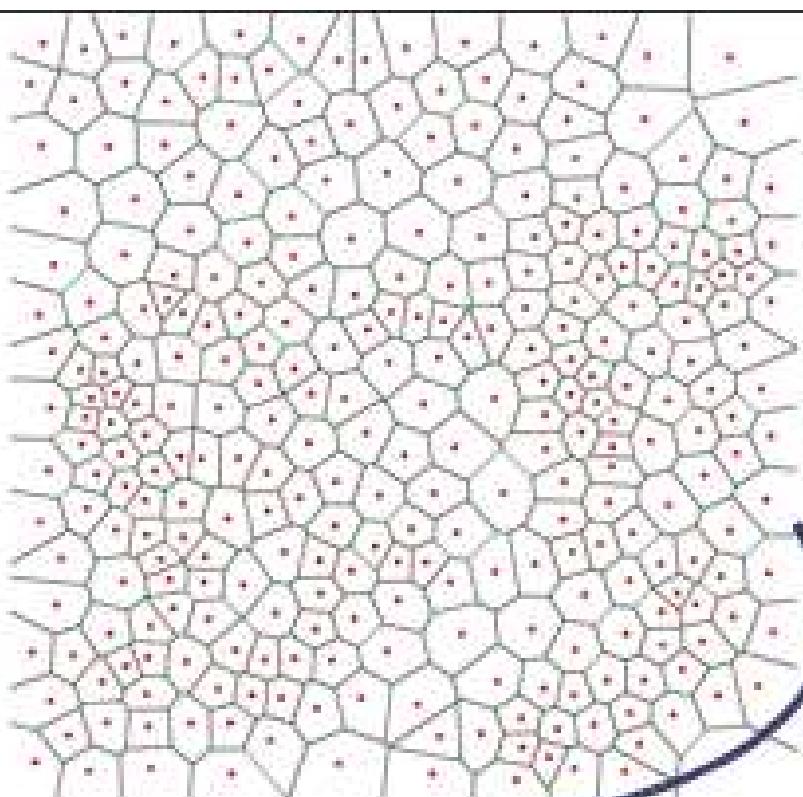
- [Polygons](#)
- [Graph Representation](#)
- [Islands](#)
- [Elevation](#)
- [Rivers](#)
- [Moisture](#)
- [Biomes](#)
- [Noisy Edges](#)
- [Demo!](#)
- [Source code](#)
- [More features](#)
- [Future work](#)
- [More reading](#)



Most procedural map generators, including some of my own previous projects, use noise functions (simplex noise, midpoint displacement, fractal, diamond-square, perlin noise, etc.) to generate a height map. I did not do that here. Instead, I used a **graph structure** to model the things directed by gameplay constraints (elevation, roads, river flow, quest locations, monster types) and **noise functions** to model the variety not constrained by gameplay (coastline shape, river placement, tree placement).

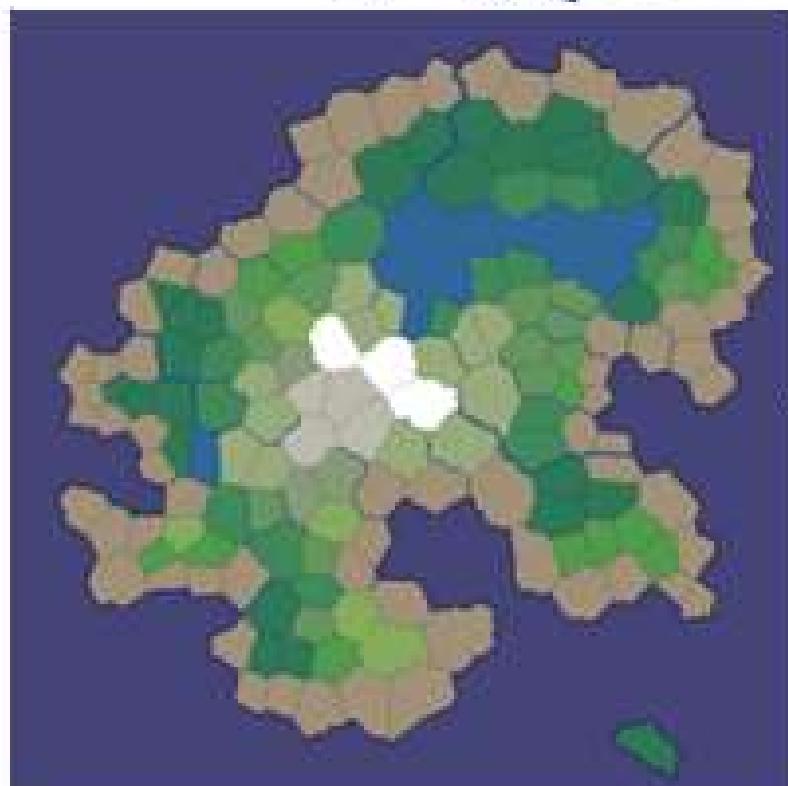
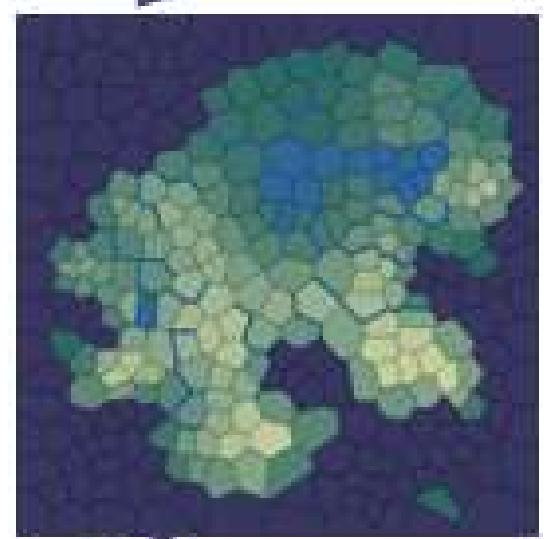
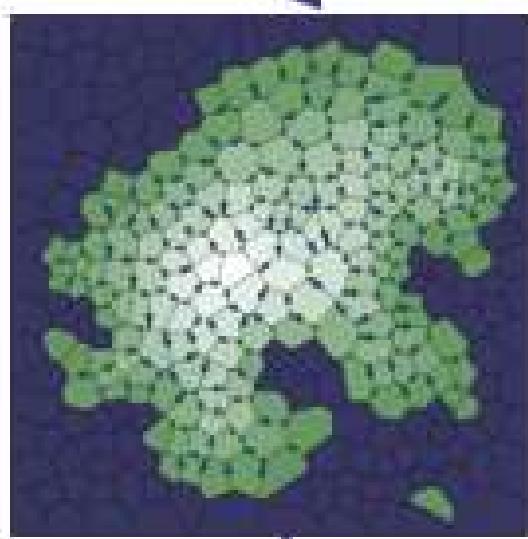
There were three main things I wanted for this project: good coastlines, mountains, and rivers. For the coastline, I wanted to make island maps that are surrounded by ocean, so that I don't have to deal with people walking to the edge of the map. For the mountains, I started with something simple: mountains are whatever's farthest from the coastline, so that you can always walk uphill to reach the top. For the rivers, I started with something simple: draw rivers from the coast to the mountains, so that you can always follow rivers down to the beach.

First, try the [Flash demo](#) (2010) or an [HTML5 demo](#)<sup>[1]</sup> (2017). Read on to learn how it works, or get the [source code](#). Here's the overview of the process:

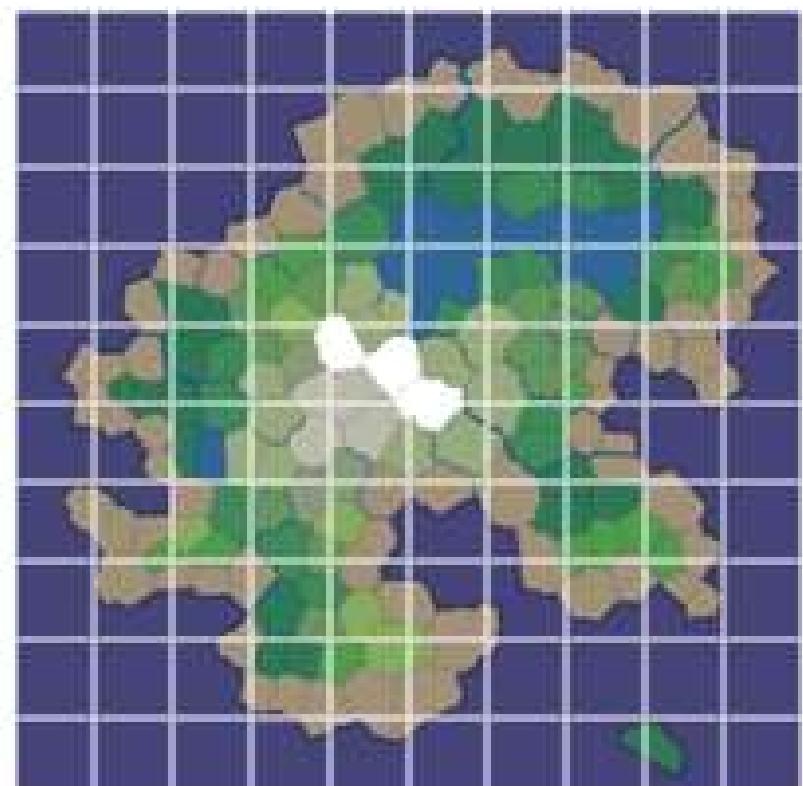


1. start with  
a graph structure  
made of polygons

2. annotate

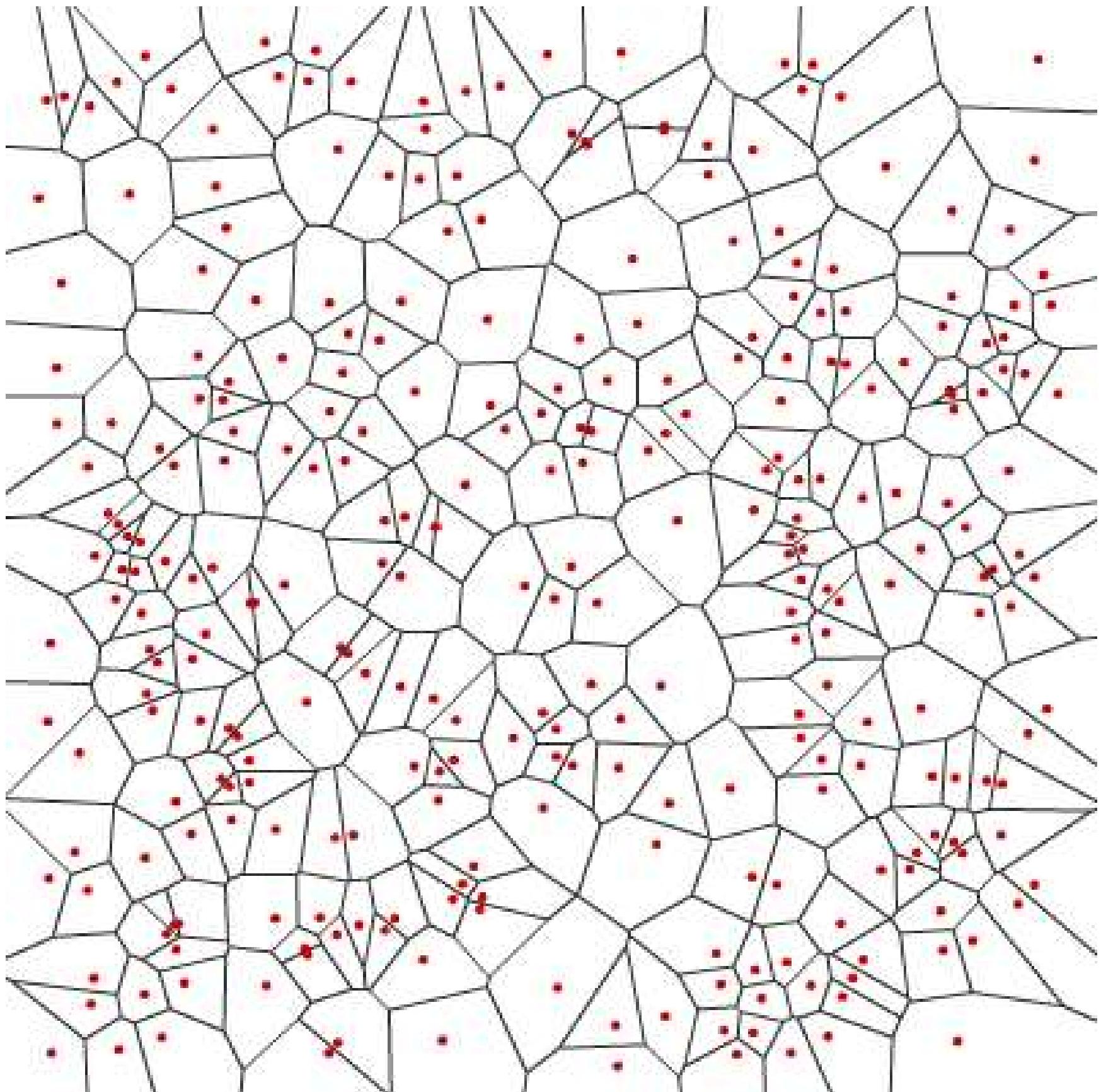


3a. output to polygons

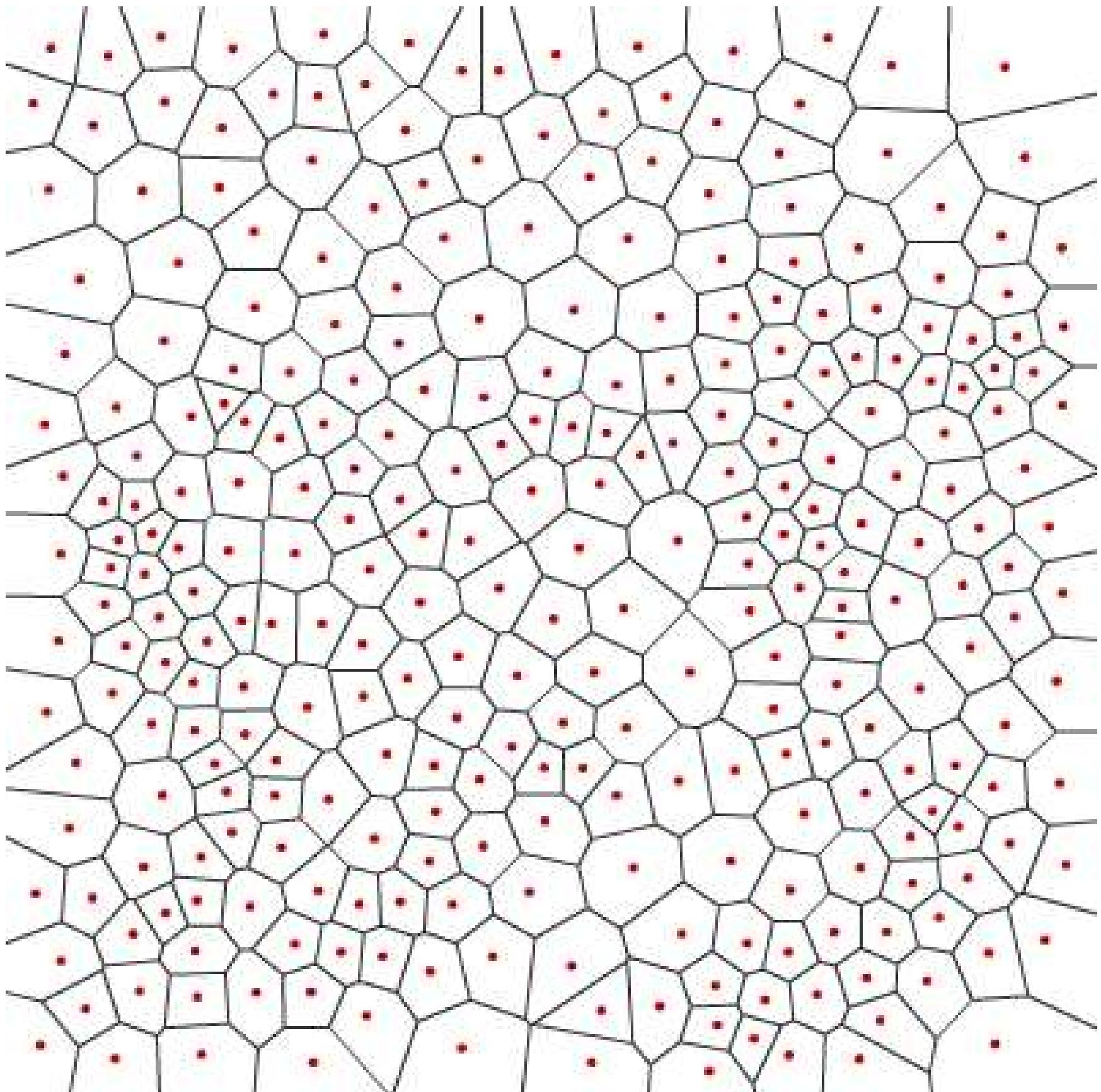


3b. optionally to tiles

Every project will have its own gameplay constraints. For this project, the gameplay constraints were partially taken from [Realm of the Mad God](#)<sup>[2]</sup>, a multiplayer RPG in which players start on the beach playing alone and then later join together on the mountaintop to fight bosses. Elevation directly corresponds to difficulty, and must monotonically increase, so that was a key constraint in the design. Elevation in Minecraft on the other hand isn't constrained the same way, so the noise function they use works for that game. In multiplayer Age of Empires, the location of resources is constrained by the need to be somewhat balanced among the players; in Minecraft the distribution of resources is not constrained. When writing your own map generator, think about what which aspects of your map are set by the design and which can vary from map to map. Each of the ideas on this page can be used separately or together in your own map generator project.



The polygon shapes and sizes are a bit irregular. Random numbers are more “clumpy” than what people expect. I want something closer to semi-random “blue noise”, or [quasirandomness](#)<sup>[8]</sup>, not random points. I approximate that by using a variant of [Lloyd relaxation](#)<sup>[10]</sup>, which is a fairly simple tweak to the random point locations to make them more evenly distributed. Lloyd relaxation replaces each point by the [centroid](#)<sup>[10]</sup> of the polygon. In my code I merely average the corners (see `improveRandomPoints`). Here’s the result after running approximate Lloyd relaxation twice:



Compare it to running [once](#) or [fifty times](#). The more iterations, the more regular the polygons get. Running it twice gives me good results but every game will vary in its needs.

Polygon sizes are improved by moving polygon centers. The same approach works to improve edge lengths. Moving corners by averaging the nearby centers produces more uniform edge lengths, although it occasionally worsens the polygon sizes. In the code, see the `improveCorners` function. However, moving corners changes it from a Voronoi diagram to a [barycentric dual mesh](#)<sup>[11]</sup>. The algorithms for this map generator work with either style. Voronoi polygons are more uniformly sized, with varying shapes; barycentric dual polygons are more uniformly shaped, and the corners are more uniformly spaced. In the rest of the article I still call them Voronoi polygons and use screenshots of Voronoi, but the final demo uses barycentric dual instead.

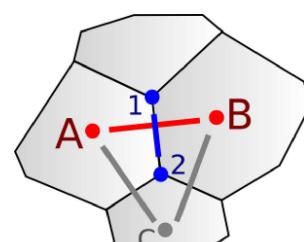
Using Voronoi adds some complexity so if you want to start with something simpler, try a square or hexagonal grid (you can see this in the demo). The rest of the techniques in this article will work with a grid. Optionally, randomly perturb the vertices of the grid to make it a little more natural looking.

---

## 2 Map Representation

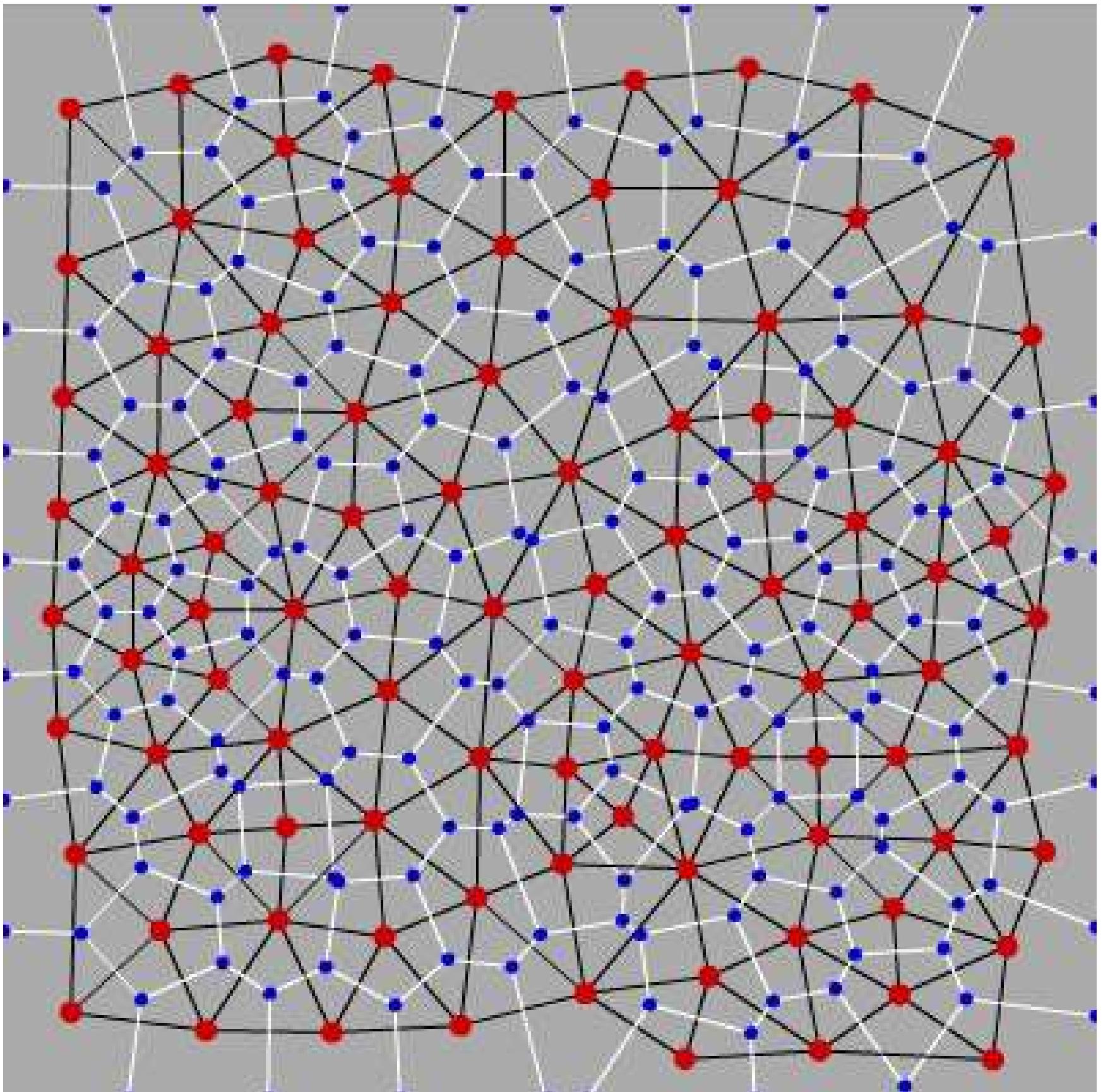
I'm representing the map as two related [graphs](#)<sup>[12]</sup>: nodes and edges. The first graph has nodes for each polygon and edges between adjacent polygons. It represents the [Delaunay triangulation](#)<sup>[13]</sup>, which is useful for anything involving adjacency (such as pathfinding). The second graph has nodes for each polygon corner and edges between corners. It contains the shapes of the polygons. It's useful for anything involving the shapes (such as rendering borders).

The two graphs are related. Every triangle in the Delaunay triangulation corresponds to a polygon corner in the Voronoi diagram. Every polygon in the Voronoi diagram corresponds to a corner of a Delaunay triangle. Every edge in the Delaunay graph corresponds to an edge in the Voronoi graph. You can see this in the following diagram:



Polygon  $A$  and  $B$  are adjacent to each other, so there is a (red) edge between  $A$  and  $B$  in the adjacency graph. For them to be adjacent there must be a polygon edge between them. The (blue) polygon edge connects corners  $1$  and  $2$  in the Voronoi shape graph. Every edge in the adjacency graph corresponds to exactly one edge in the shape graph.

In the Delaunay triangulation, triangle  $A-B-C$  connects the three polygons, and can be represented by corner  $2$ . Thus, corners in the Delaunay triangulation are polygons in the Voronoi diagram, and vice versa. Here's a larger example showing the relationship, with Voronoi polygon centers in red and corners in blue, and the Voronoi edges in white and the Delaunay triangulation in black:



This duality means that I can represent the two graphs together. There are [several approaches<sup>\[14\]</sup>](#) for combining the data from the two graphs. In particular, [edges can be shared<sup>\[15\]</sup>](#). Each edge in a normal graph points to two nodes. Instead of representing two edges in the two graph separately, I made edges point to *four* nodes: two polygon centers and two corners. It turns out to be quite useful to connect the two graphs together.

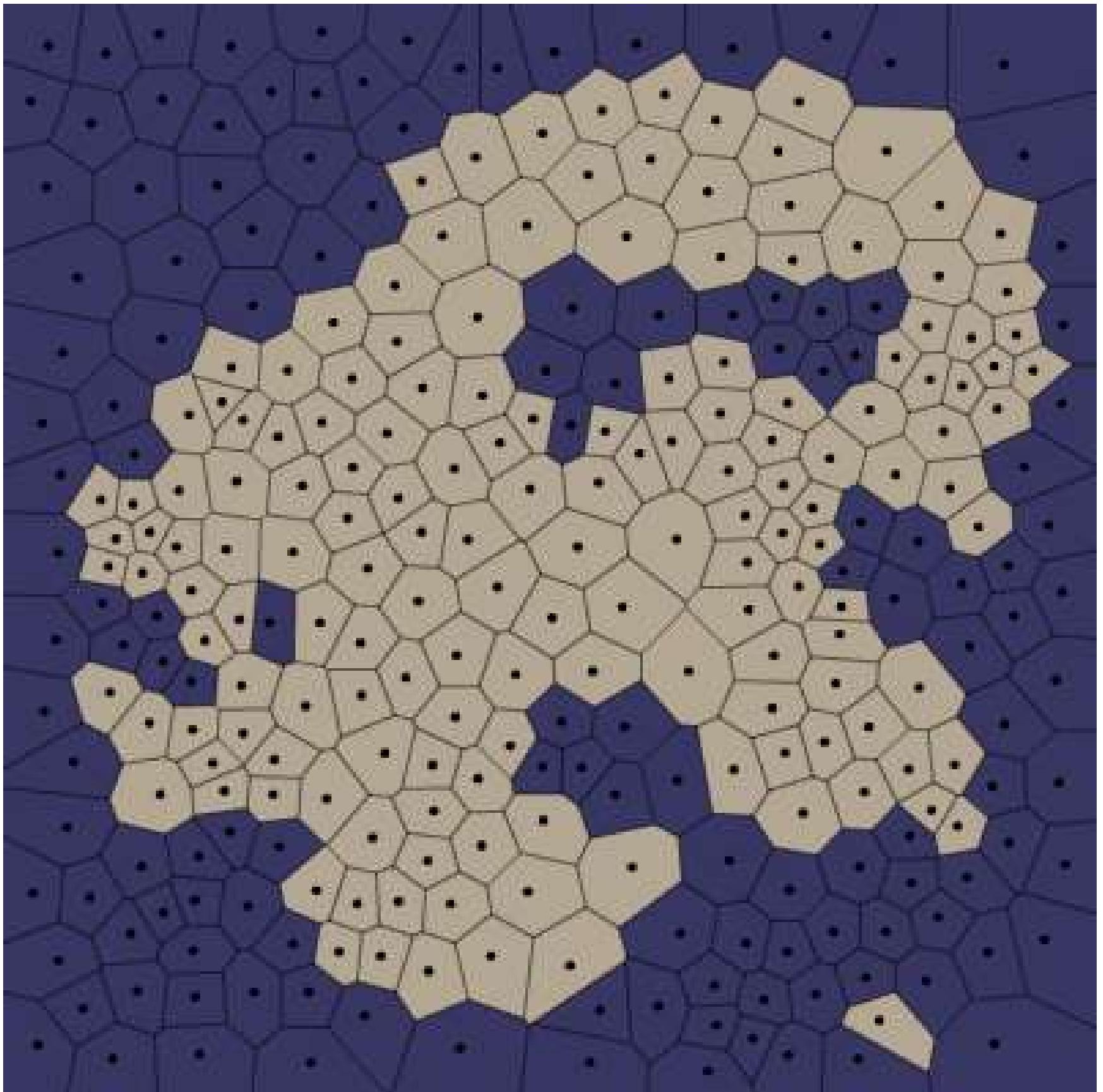
With the combined representation, I can now use the Relationships Between Grid Parts sections of my [article on grids<sup>\[16\]</sup>](#). They're not grids so I'm not assigning grid coordinates, but many of the algorithms that work on grids also work here, and the algorithms that work on graphs also work here (on either of the two graphs).

In the code, the `graph/` directory has three classes: `Center`, `Corner`, and `Edge`:

- `Center.neighbors` is a set of adjacent polygons
- `Center.borders` is a set of bordering edges
- `Center.corners` is a set of polygon corners
- `Edge.d0` and `Edge.d1` are the polygons connected by the Delaunay edge
- `Edge.v0` and `Edge.v1` are the corners connected by the Voronoi edge
- `Corner.touches` is a set of polygons touching this corner
- `Corner.protrudes` is a set of edges touching the corner
- `Corner.adjacent` is a set of corners connected to this one

The second step is to draw the coastline. The borders of the map need to be water, but you can mark the other polygons as either water or land, using any approach you want. The coastline is then all the edges where land and water meet.

Here's an example that divides the world into land and water:



In the code, `Map.as` contains the core map generation code. The `IslandFunction` returns `True` if a position is land, and `False` for water. There are four island functions included in the demo:

- `Radial` uses sine waves to produce a round island
- `Perlin` uses Flash's built-in Perlin noise to control the shape
- `Square` fills the entire map with land
- `Blob` draws my [blob logo](#)<sup>[17]</sup>

You can use any shape, (including [pizza box stains](#)<sup>[18]</sup> or [clouds](#)<sup>[19]</sup> or a [torn up sheet of paper](#)<sup>[20]</sup>). Or let the game designer draw their own shape, which is what I implemented in [mapgen4](#)<sup>[21]</sup>.

The code assigns water/land to both polygon centers and corners:

1. Assign water/land to the `corners` by setting `Corner.water` based on the `IslandFunction`.
2. Assign water/land to the `polygons` by setting `Center.water` if some fraction of the corners have `water` set.

A simple flood fill starting from the border of the map can determine which water areas are oceans (connected to the border) and lakes (surrounded by land):



In the code, the flood fill runs on the polygon centers, and then we can decide what happens to corners:

1. Set `Center.ocean` for any polygon connected to the borders of the map through water polygons. If `Center.water` is set but `.ocean` is not, then it's a lake.
2. Set `Center.coast` if the polygon is land but has an ocean border. Coastal areas will later get drawn as beaches.
3. Set `Corner.ocean` if the corner is surrounded by ocean polygons.
4. Set `Corner.coast` if the corner touches ocean and land polygons.
5. Reset `Corner.water` to be consistent with the surrounding area.

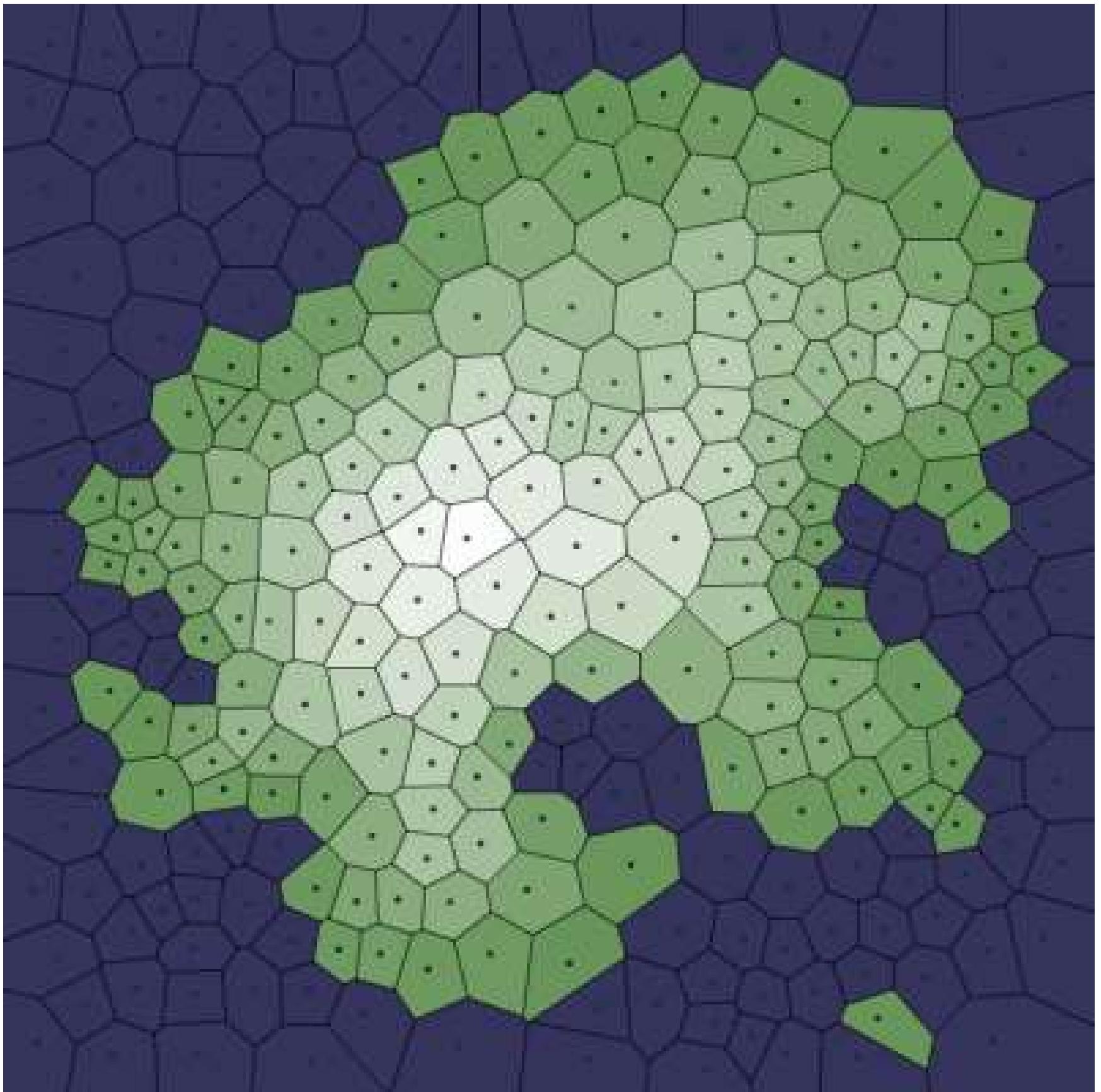
---

4      **Elevation**

#

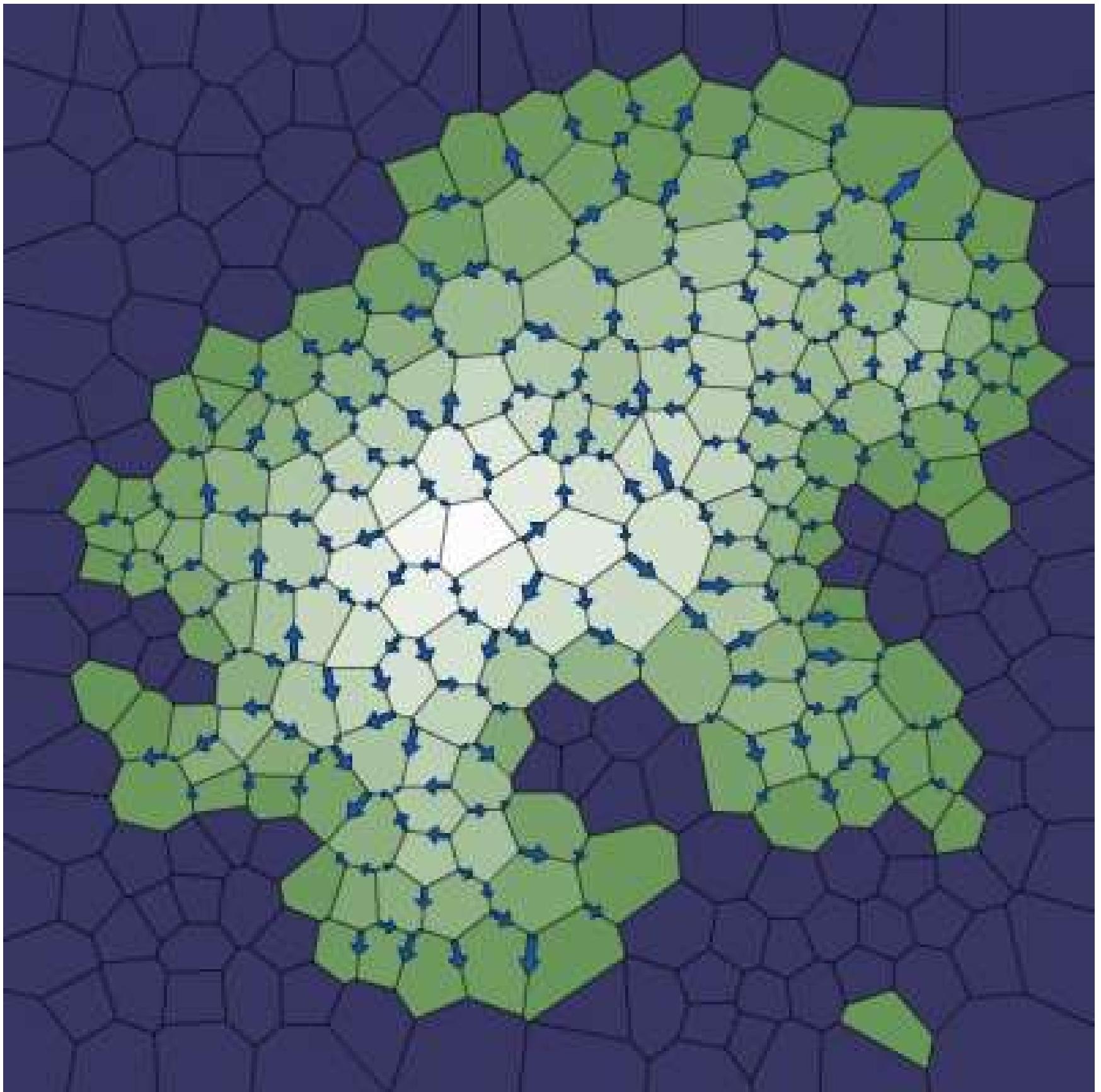
The most realistic approach would have been to define elevation first, and then define the coastline to be where the elevation reaches sea level. Instead, I'm starting with the goal, which is a good coastline, and working backwards from there. I set elevation to be the **distance from the coast**. I originally tried elevations at polygon centers but setting elevations at corners worked out better. Corner-to-corner edges can serve as ridges and valleys. After calculating the elevation of corners (`Corner.elevation`), the polygon elevation (`center.elevation`) is the average of the elevation at the corners. See the functions `Map.assignCornerElevations` and `Map.assignPolygonElevations`.

Water polygons don't count towards the distance. This is both because I expect lakes to be flat instead of sloped, and because this tends to build valleys around lakes, which helps guide rivers towards lakes.



One problem with the simple definition is that some islands have too many mountains and others have too few. To fix this, I redistribute the elevations to match a desired distribution, which has more low elevation land (coastline) than high elevation land (mountains). First, I sort the corners by elevation, and then I reset the elevation  $x$  of each to match the inverse of the desired cumulative distribution:  $y(x) = 1 - (1-x)^2$ . In the `Map.redistributeElevations` function,  $y$  is the position in the sorted list, and  $x$  is the desired elevation. Using the quadratic formula, I can solve for  $x$ . This preserves ordering so that elevations always increase from the coast to the mountains.

For any location, going downhill will eventually lead to the ocean. This diagram shows the steepest downhill direction from every corner, stored in `Corner.downslope`:



By following the downhill arrows from any location, we eventually reach the ocean. This will be useful for rivers but may also be useful for calculating [watersheds](#)<sup>[22]</sup> and other features.

I had two main goals for elevation:

1. **Biome types:** high elevations get snow, rock, tundra; medium elevations get shrubs, deserts, forests, and grassland; low elevations get rain forests, grassland, and beaches.
2. **Rivers** flow from high elevations down to the coast. Having elevations that always increase away from the coast means that there's no local minima that complicate river generation.

In addition, games may define their own use of elevation data. For example, [Realm of the Mad God](#)<sup>[23]</sup> uses elevation to distribute monsters.

This elevation calculation works for simple islands, which is what I needed for Realm of the Mad God. For continent generation, you'll want to change this step to generate one or more mountain ranges that aren't necessarily in the center, as well as isolated volcanos.

---

5

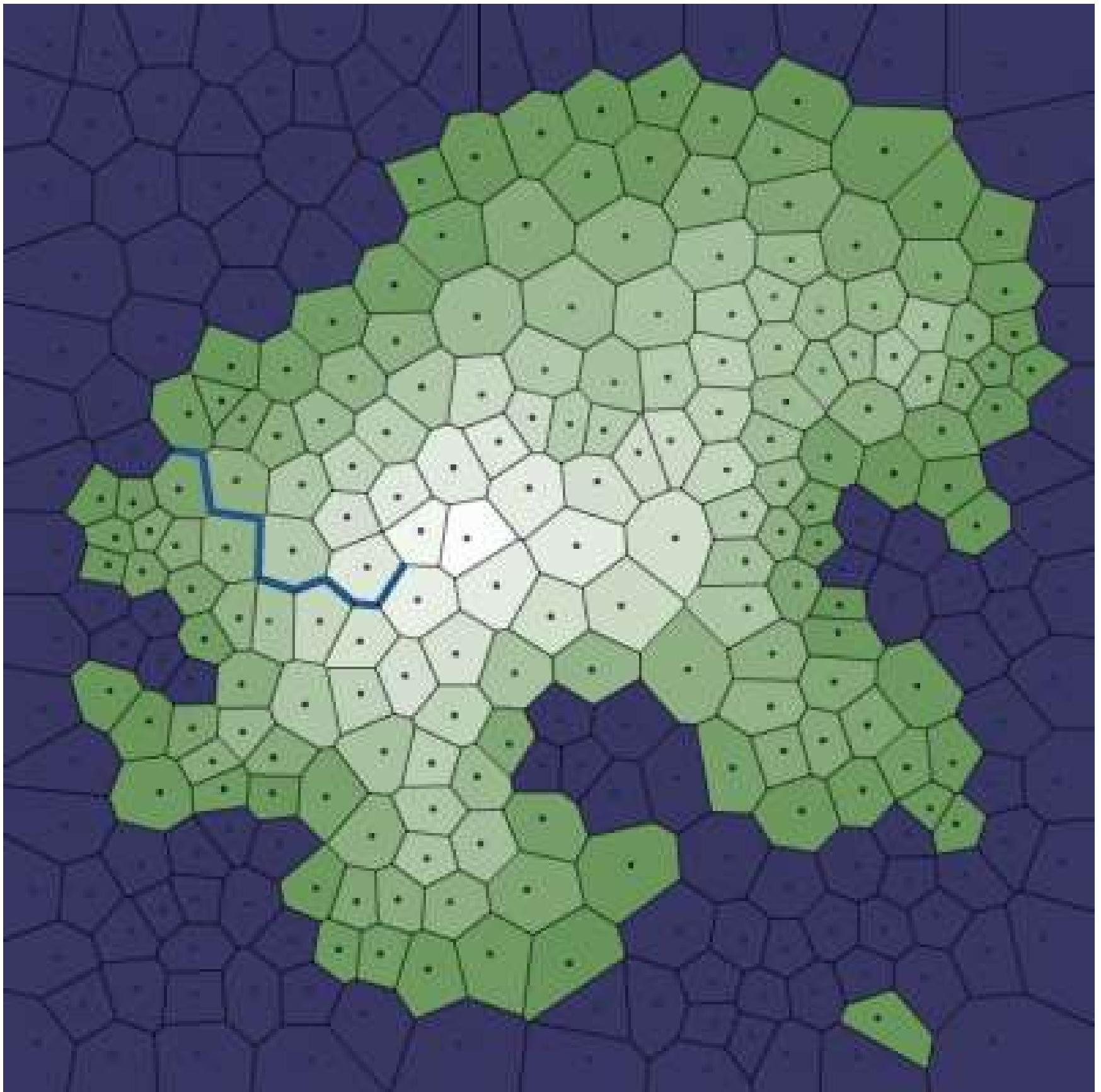
## Rivers

#

Rivers and lakes are the two fresh water features I wanted. The most realistic approach would be to define moisture with wind, clouds, humidity, and rainfall, and then define the rivers and lakes based on where it rains. Instead, I'm starting with the goal, which is good rivers, and working backwards from there.

The island shape determines which areas are water and which are land. Lakes are water polygons that aren't oceans.

Rivers use the downhill directions shown earlier. I choose random corner locations in the mountains, and then follow the `Corner.downslope` path down to the ocean. The rivers flow from corner to corner:



I tried both polygon centers and corners, but found that the corner graph made for much nicer looking rivers. Also, by keeping lakes flat, elevation tends to be lower near lakes, so rivers naturally flow into and out of lakes. Multiple rivers can share the lower portion of their path. Every time a river flows through an edge, I increase the water volume stored in `Edge.river` by 1. At rendering time, the river width is the square root of the volume. This approach is simple and works well.

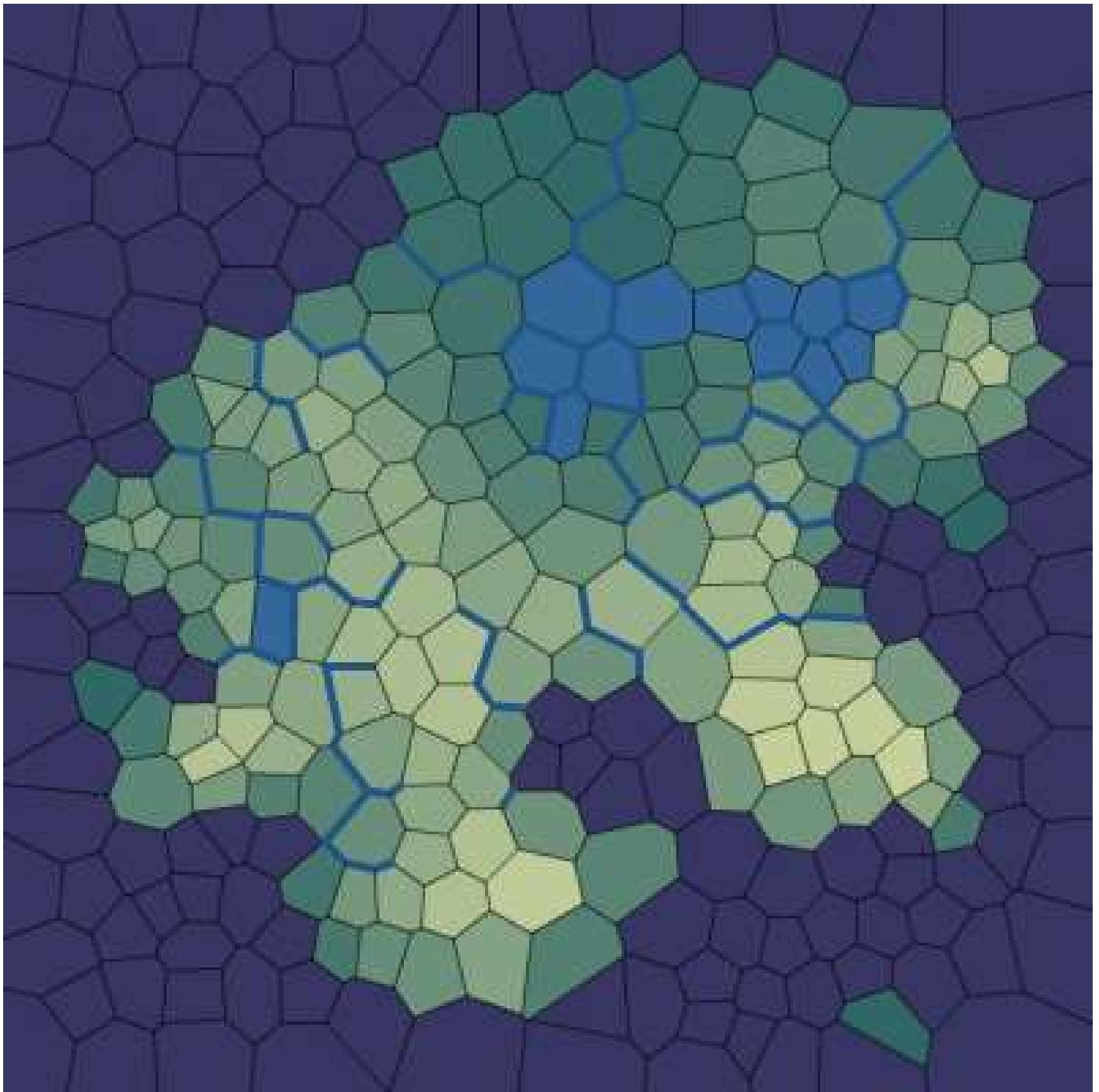
---

6      **Moisture**

---

#

Since I'm working backwards, I don't need moisture to form rivers. However, moisture would be useful for defining **biomes** (deserts, swamps, forests, etc.). Since rivers and lakes should form in areas with high moisture, I defined moisture to decrease as **distance from fresh water increases**. `Corner.moisture` is set to  $a^k$  for some  $a < 1$  (e.g. 0.95), and  $k$  being the distance. There are unfortunately some tuning parameters in `Map.assignCornerMoisture` that I tweaked until the maps looked reasonable:



As with elevation, I redistribute moisture to match a desired distribution. In this case, I want roughly equal numbers of dry and wet regions. The desired cumulative distribution is  $y(x) = x$ , so the redistribution code is very simple. I sort by moisture and then assign the moisture of each corner to that corner's position in the sorted list. See `Map.redistributeMoisture` for the code.

In this map generator, moisture is only used for biomes. However, games may find other uses for the moisture data. For example, [Realm of the Mad God](#)<sup>[24]</sup> uses moisture and elevation to distribute vegetation and monsters.

## 7

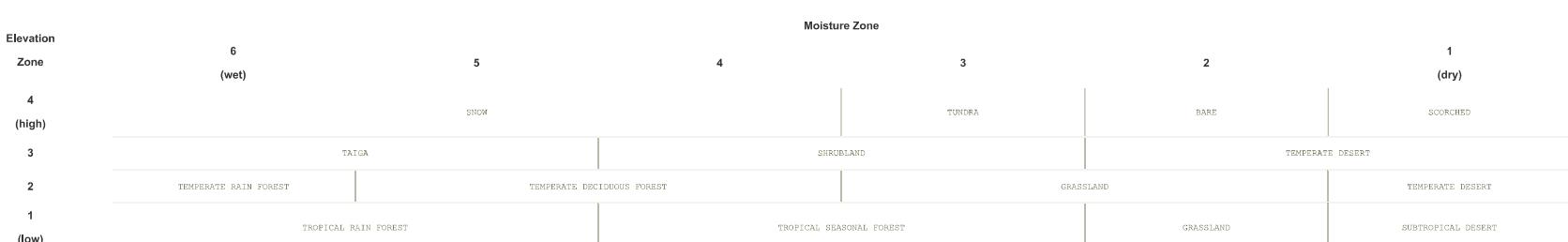
## Biomes

#

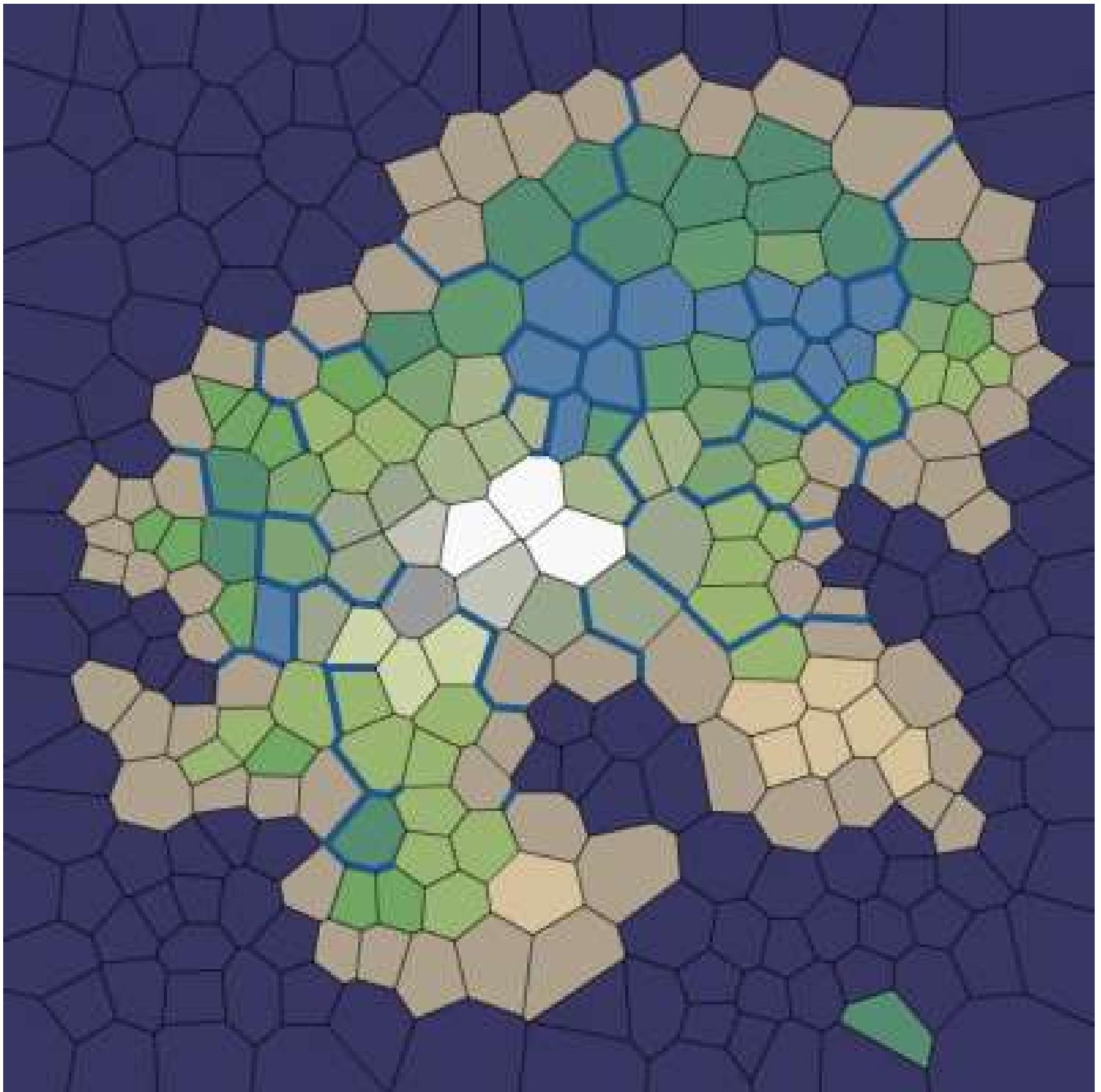
Together, elevation and moisture provide a good amount of variety to define biome types. I use elevation as a proxy for temperature. If this were a continent generator, latitude might be a contributor to temperature. Also, wind, evaporation, and rain shadows might be useful for transporting moisture as humidity. However, for this generator I kept it simple. Biomes first depend on whether it's water or land:

- `OCEAN` is any water polygon connected to the map border
- `LAKE` is any water polygon not connected to the map border, or `ICE` lake if the lake is at high elevation (low temperature), or `MARSH` if it's at low elevation
- `BEACH` is any land polygon next to an ocean

For all land polygons, I started with the [Whittaker diagram](#)<sup>[25]</sup> and adapted it to my needs:



Here's the result:



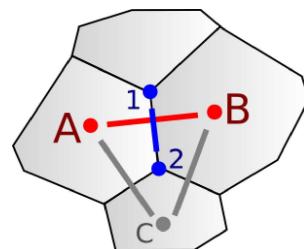
These biomes look good in the map generation demo, but each game will have its own needs. [Realm of the Mad God](#)<sup>[26]</sup> for example ignores these biomes and uses its own (based on elevation and moisture).

---

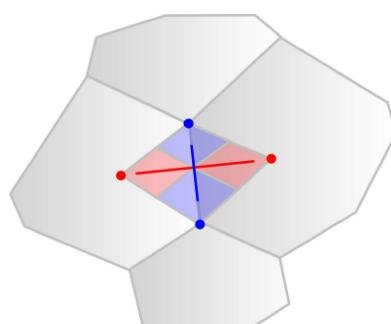
## 8 Noisy Edges #

For some games, the polygonal maps are sufficient. However, in other games I want to hide the polygon structure. The main way I do that is to replace the polygon borders with a noisy line. Why would I want a polygon structure if I'm going to hide it? I think game mechanics and pathfinding benefit from the underlying structure.

Recall from earlier that there are *two* graphs: one for Voronoi corners ( $1, 2$  in the diagram below) and edges (blue lines), and one for polycon centers ( $A, B$ ) and Delaunay edges (red lines) between them:

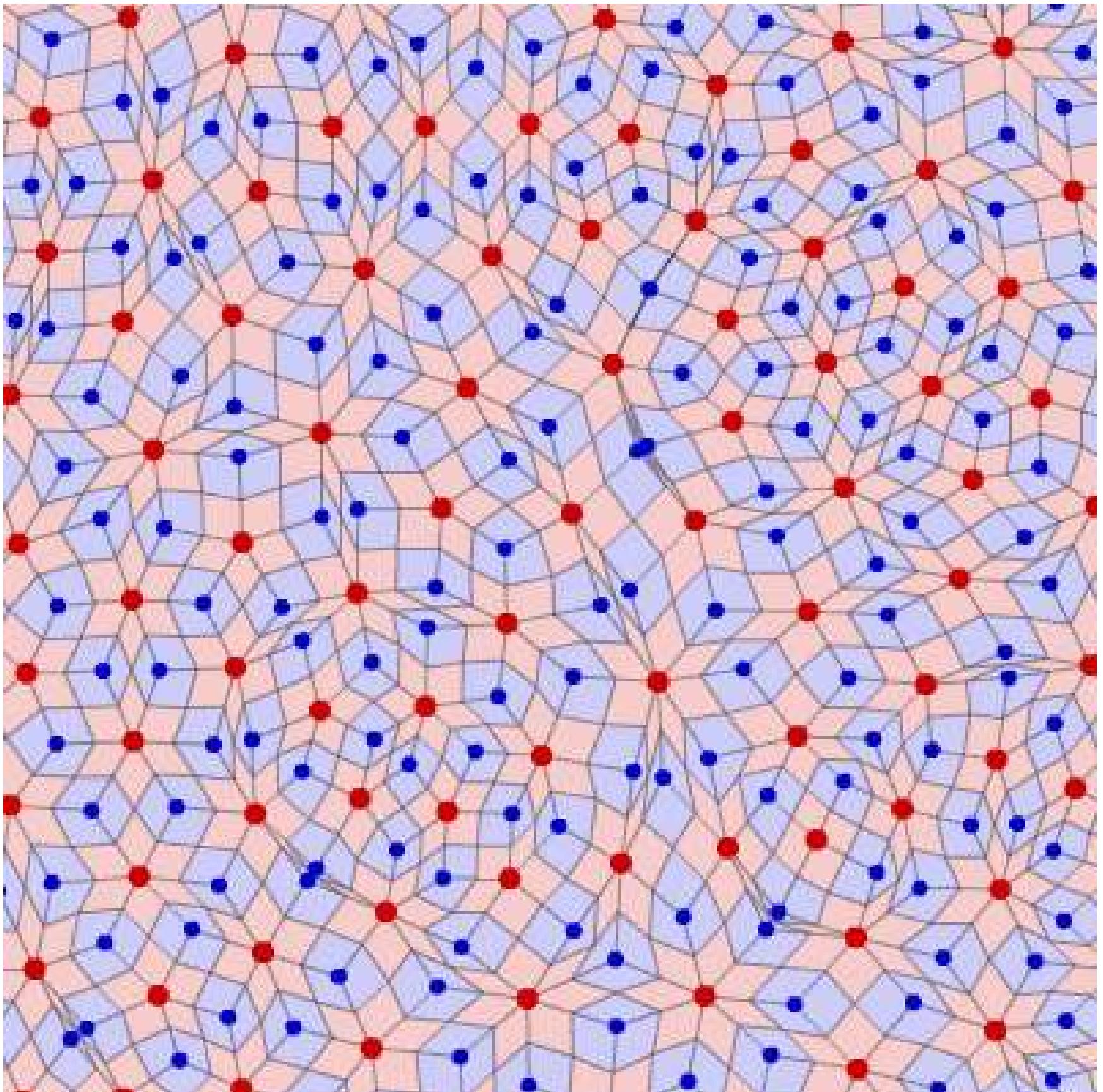


I wanted to make both types of line noisy without making them cross lines from other polygons. I also wanted to make them as noisy as feasible. I realized that points  $A, 1, B$ , and  $2$  form a quadrilateral, and I could constrain the wanderings of the line segment to that quadrilateral:



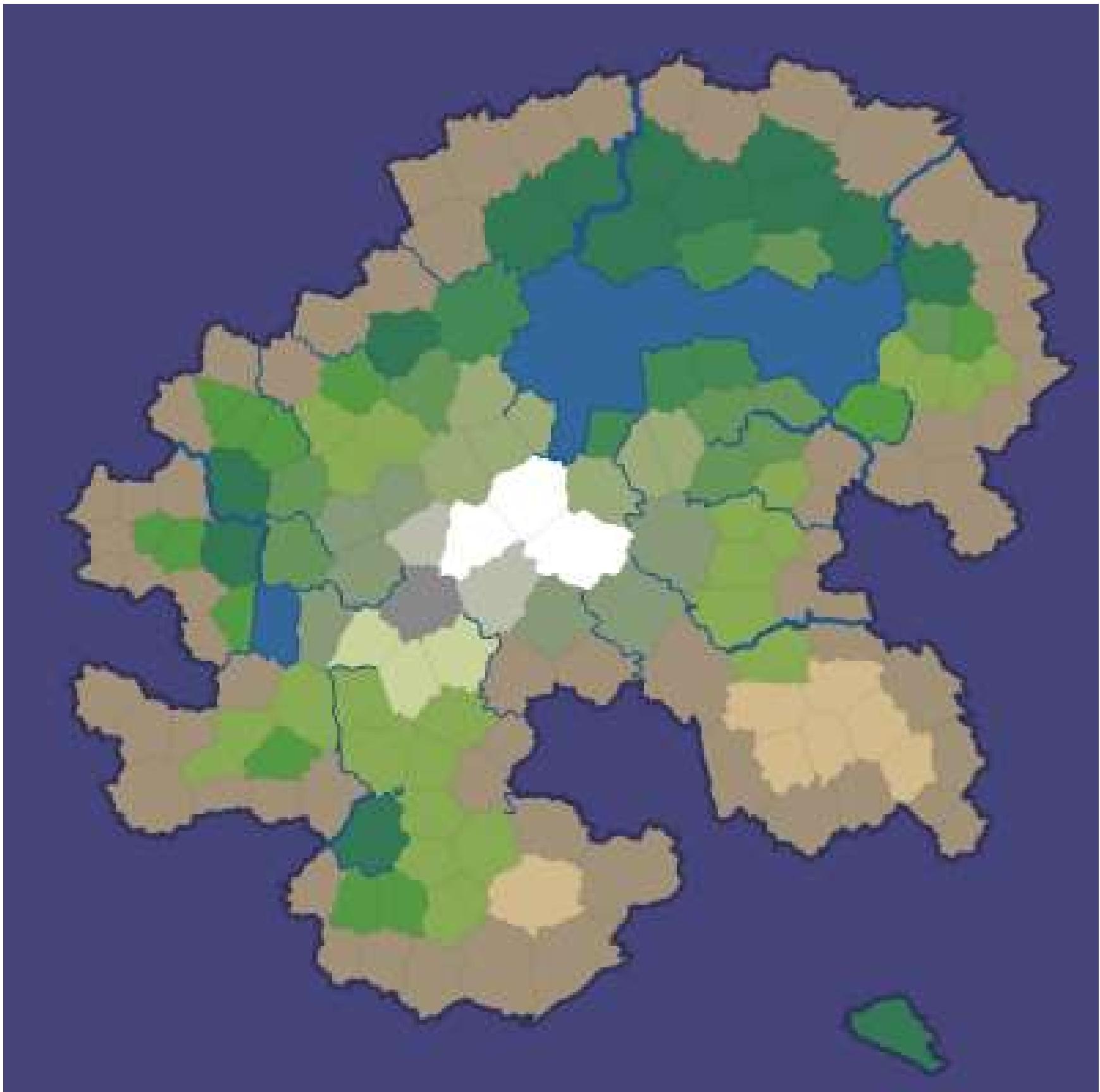
I further divided the quadrilateral into four quadrilaterals. Two were usable for the red (Delaunay) edge and two for the blue (Voronoi) edge. As long as the lines stayed within their allocated space and met in the center, they'd never cross each other. That takes care of constraining them. Note that the quadrilateral may not be convex; to divide it properly, I divide it at the midpoint of the Voronoi edge instead of at the intersection of the Voronoi and Delaunay edges.

The entire map can be divided up into these quadrilateral regions, with no space left over:



That ensures that the noisy lines aren't constrained any more than necessary. (I wonder if these quadrilaterals would be useful for game mechanics.)

I can use any noisy line algorithm that fits within these constraints. I decided to subdivide the quadrilaterals recursively and stitch line segments together within the small quadrilaterals into a complete edge. The algorithm is in `NoisyEdges.as`, in `buildNoisyLineSegments`. The result is that the polygon edges are no longer straight:

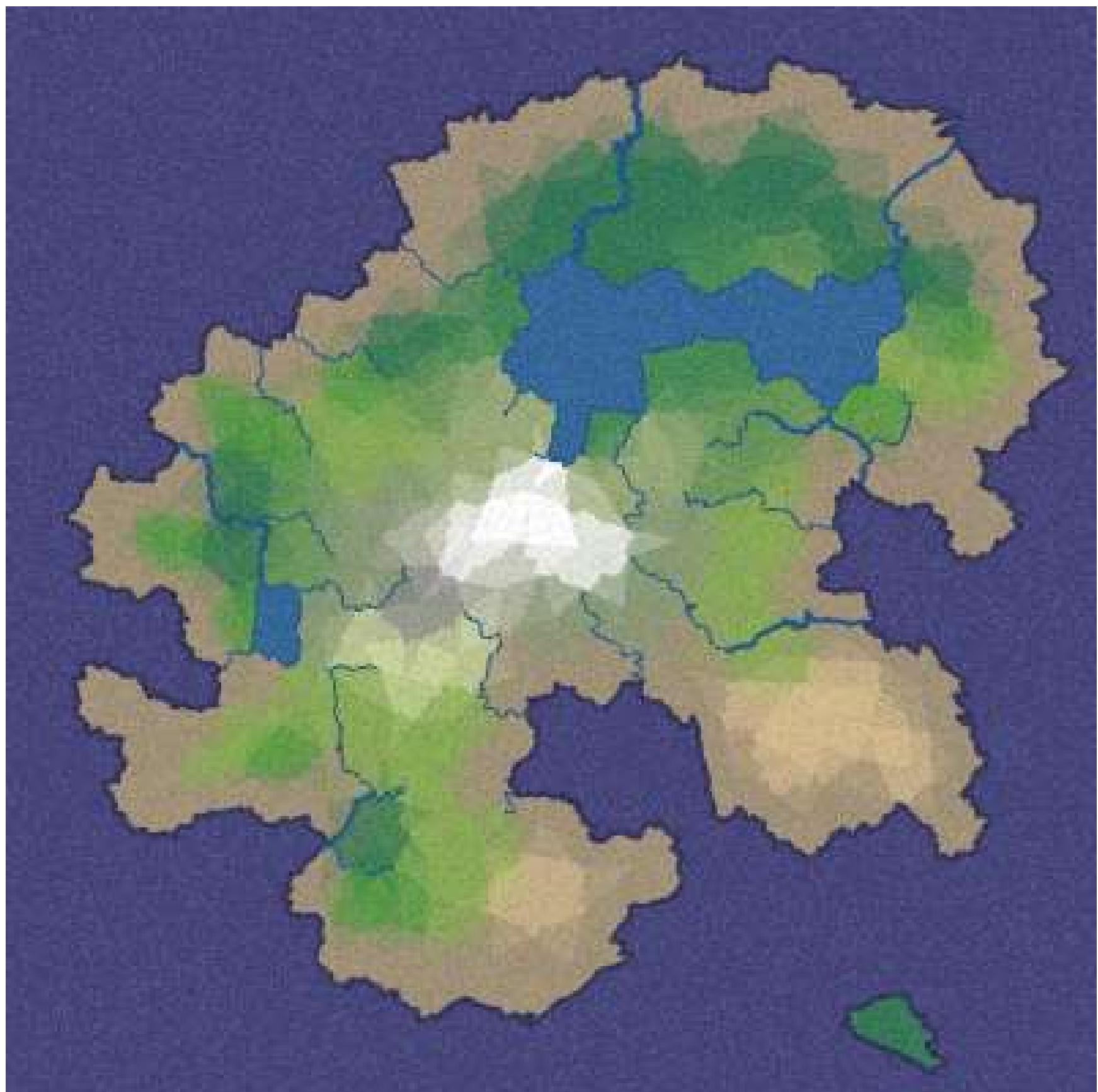


There are three places to tune the noisiness:

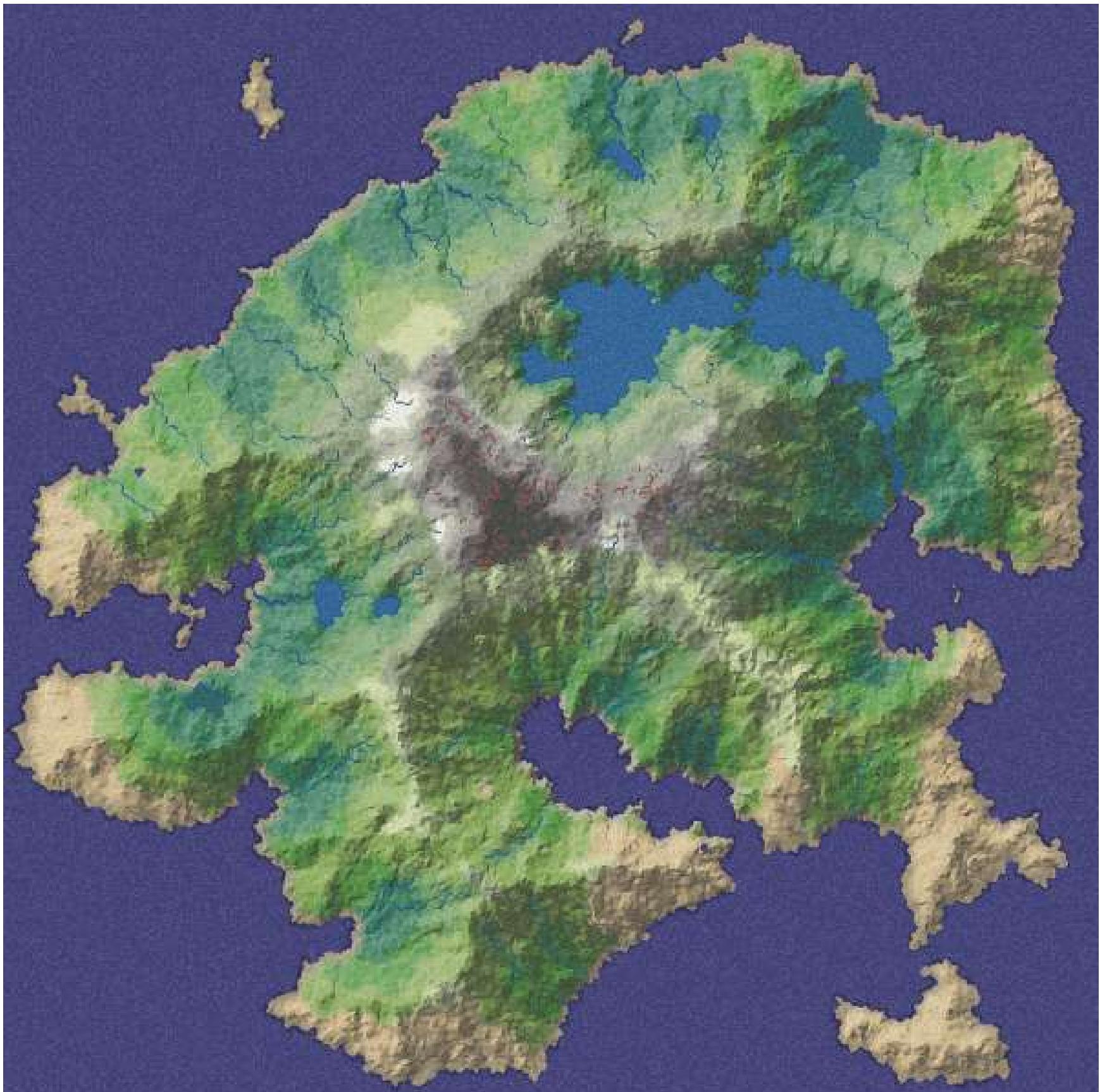
1. The recursive function ends when the segments are shorter than some length. I have examples at [segment size 7](#), [segment size 4](#), and [segment size 1](#). In the map demo I use segment size 1 for rivers and coastlines, 3 where biomes meet, and 10 elsewhere.
2. There's a tradeoff between how much of the space goes to the red quadrilaterals (Delaunay edges) and blue quadrilaterals (Voronoi edges). I set `NoisyEdges.NOISY_LINE_TRADEOFF` to 0.5.
3. There's a range of random numbers in `NoisyEdges.subdivide`. In the current demo it's from 0.2-0.8, but it can be up to 0.0-1.0. Also, the random numbers don't have to be linearly chosen. More visual noise results if you avoid the space around 0.5.

Noisy edges turn out to have a large impact on the map appearance, especially for rivers and coastlines. For newer projects I've used a slightly different algorithm described [here](#)<sup>[27]</sup>.

I'm generally a fan of [noise in game art](#)<sup>[28]</sup>, and wanted to add a little bit of noise to these maps as well. In a real game map the noise might reflect vegetation or small variations in terrain. In the demo (`mapgen2.as`) I filled the screen with a random noise texture by adding a noise bitmap on top. I also smoothed the borders between adjacent polygons by blending the colors in stages:



Here's a rendering with 16,000 polygons, noisy edges, a noise texture overlay, and simple lighting:

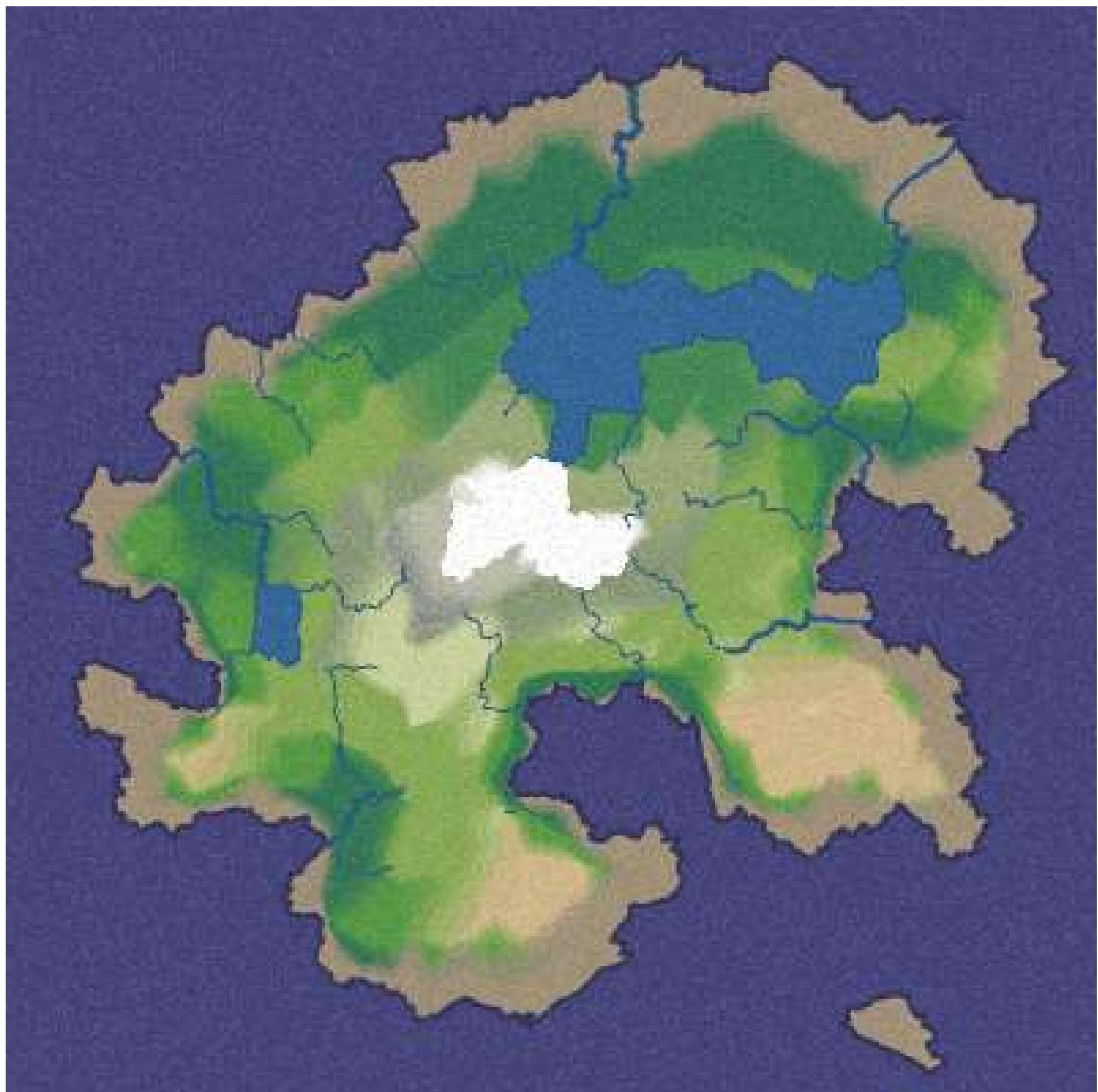


---

10    [Smooth biome transitions](#)

#

A different way of blending the biomes at polygon boundaries is to build gradients using the elevation and moisture at each *corner*, and then assigning biomes per pixel:



If the game doesn't need an entire polygon to be the same biome, this approach can be useful for making more interesting boundaries.

---

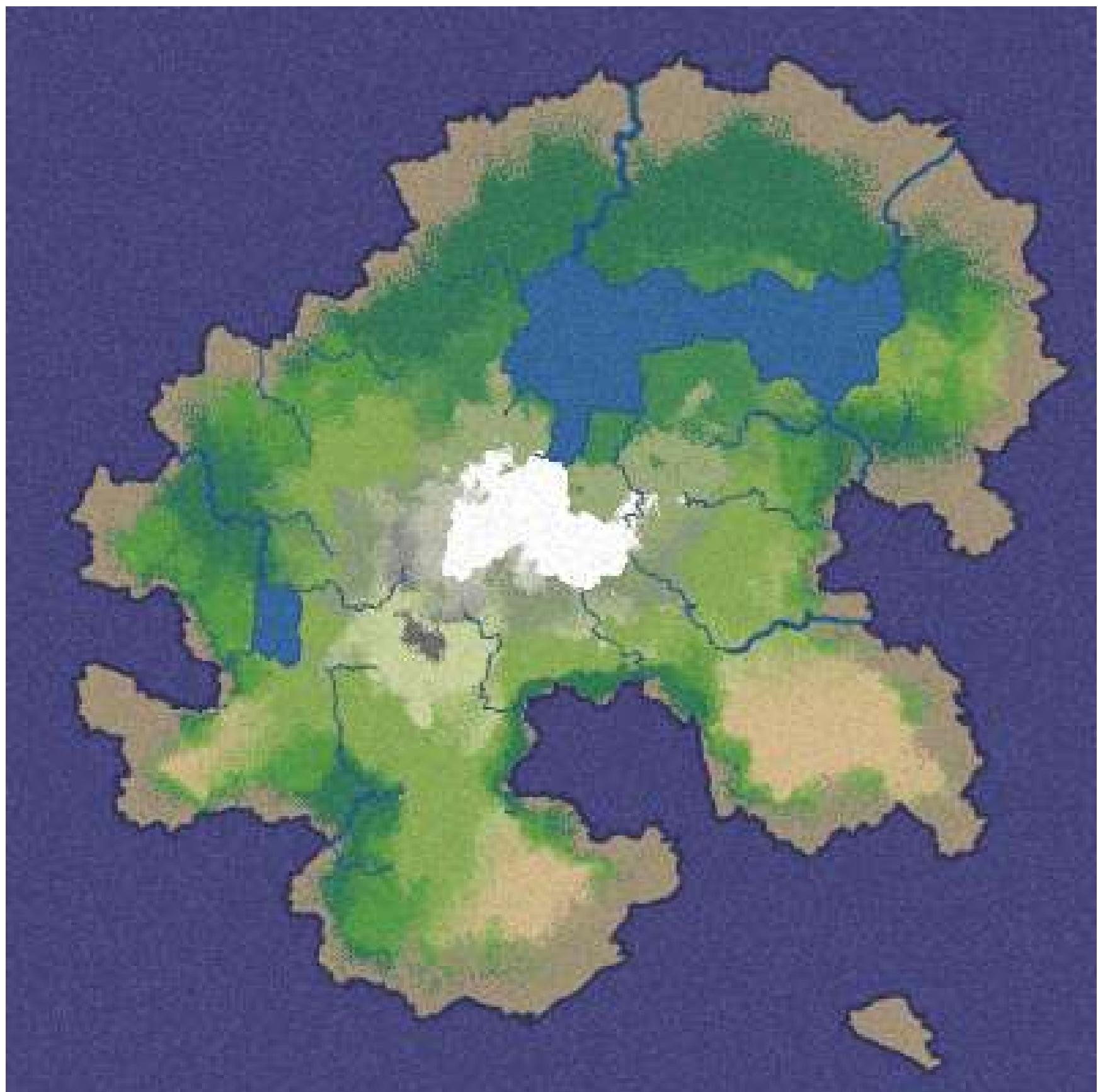
11      **Distorted biome transitions**

#

Another way to make the map look less polygon-like is to distort the elevation and moisture maps:

1. Add Simplex or random noise to the elevation and moisture at each pixel.
2. Sample nearby points using Simplex or random noise to change the coordinate.

Here's an example of what this can do:



Adding noise to the elevation and moisture will produce "dithering" in the zones near transitions. Sampling nearby points using noise will distort the shapes of the boundaries.

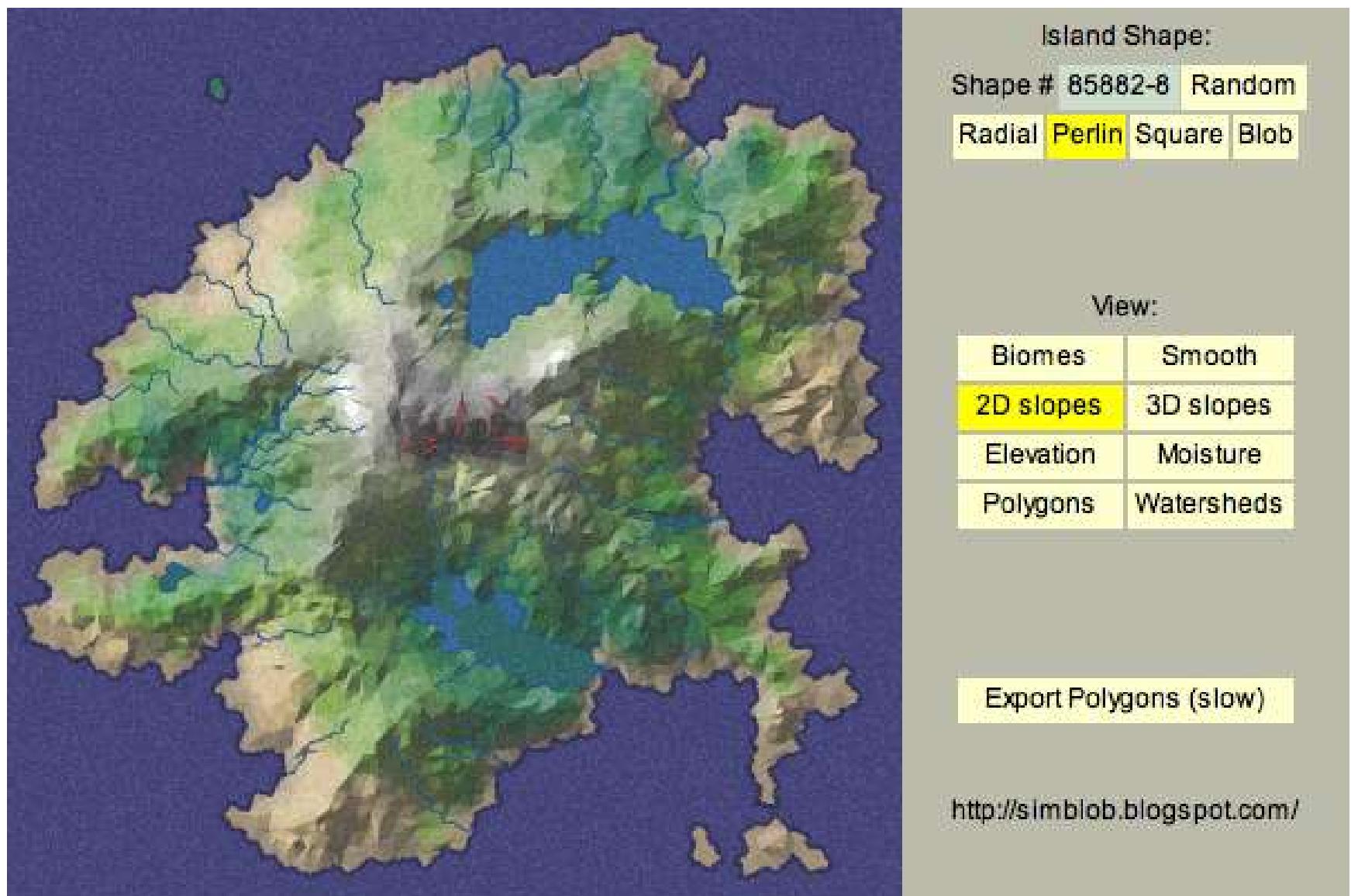
---

12

[Demo](#)

#

I wrote a Flash demo to explore the generated maps:



[Try the demo!](#)

## 13     [Source](#)

#

I've placed the source code under the MIT license: [Actionscript/Flash source code](#)<sup>[29]</sup>. If you can read Java or Javascript, I think you'll have no trouble reading the Actionscript. I don't expect that the code will be immediately useful to anyone, but it might be a useful starting point if you'd like to use these techniques for making your own game maps.

- `Map.as` is the core map generation system
- `graph/*.as` is the representation of graphs (polygons, edges, corners)
- `mapgen2.as` is the demo, with rendering and GUI
- `Roads.as` is a module adding roads along contour lines
- `Lava.as` is a module adding lava fissures to high elevation edges
- `NoisyEdges.as` is used by the demo to build noisy edges

The diagrams on this page are built with 300 polygons, the demo uses 2000 by default, and allows up to 8000. Some of the code for producing diagrams isn't checked in because it was quick and dirty code only for the diagrams on this page, and not generally useful.

If you find the code or ideas useful, I'd love to hear about it!

I've written a [tutorial that takes you through generating a Voronoi map step-by-step](#)<sup>[30]</sup> with Javascript code, including rendering. It's a simplified version of the algorithms on this page, and it should be a good starting point. I also have the slightly fancier algorithms from this page available under the Apache v2 license: [Javascript source code](#)<sup>[31]</sup>. The web demo is [here](#)<sup>[32]</sup>.

### 13.1. Projects that explore different algorithms

- Andy Gainey experimented with Voronoi diagrams on a sphere and decided to instead [build a map with subdivided isocahedrons](#)<sup>[33]</sup>, with tectonic plates, air currents, temperature, moisture, and biomes. Several people asked me how to extend my map generator to handle continents; take a look at this project and [demo](#)<sup>[34]</sup>.
- Martin O'Leary's [beautifully explained map generator](#)<sup>[35]</sup> uses Voronoi diagrams with a different terrain generator, erosion simulation, a stylized renderer, city/region generation, a name generator, and a label placement algorithm; you can run it online [here](#)<sup>[36]</sup>.
- Scott Turner<sup>[37]</sup> has explored [shading](#)<sup>[38]</sup>, contour lines, [wind patterns](#)<sup>[39]</sup>, and procedural mountain/hill icons on his blog.
- Azgaard<sup>[40]</sup> has built an [amazing vector map generator](#)<sup>[41]</sup> with the ideas from Scott and Martin, and has a great [blog](#)<sup>[42]</sup> with writeups on [biomes](#)<sup>[43]</sup>, [confluences](#)<sup>[44]</sup>, and other topics. [Open source](#)<sup>[45]</sup>.
- (2018) [Mapgen4](#)<sup>[46]</sup> is my successor to mapgen2, with different algorithms that run much faster at a much larger scale. The demo runs with 25k polygons but I've also tried it with 1M polygons, which only takes a few seconds. The same core algorithms work with [spheres](#)<sup>[47]</sup> and plate tectonics.
- Miguel Cepero of [Voxel Farm](#)<sup>[48]</sup> was inspired to make a political landscape map using Voronoi regions, and also [a map generator using tectonic plates](#)<sup>[49]</sup> with an alternative to Voronoi.
- Ryan Guy has a [project based on Martin's](#)<sup>[50]</sup>, with some nice diagrams explaining the process.
- Tommy Waters explores [plate tectonics](#)<sup>[51]</sup>, in which he shows how he's producing continents, not only islands, so the mountain ranges are not always in the center of the land masses.
- ylcrcronlth explores [moisture calculation based on a random walk](#)<sup>[52]</sup>
- Cory Lee has a [political map generator](#)<sup>[53]</sup> using Voronoi regions.
- Jesse Morgan has used ideas from this map generator to build a [city generator](#)<sup>[54]</sup> ([project page](#)<sup>[55]</sup> and [source code](#)<sup>[56]</sup>)
- Sword & Scroll uses a [Voronoi map generator to create political zones](#)<sup>[57]</sup> (baronies)
- Phill Spiess also uses Voronoi and noisy edges for [his project](#)<sup>[58]</sup> but no details are posted.
- Christophe Le Besnerais has a [Javascript version](#)<sup>[59]</sup> that has a more advanced water flow model; luckylooke has a version of it [for Phaser.io](#)<sup>[60]</sup>.
- Kaelan Cooter has a [hexagon world map generator](#)<sup>[61]</sup> with elevation, moisture, biomes, rainfall, aquifers, and territories.
- Andy Lo<sup>[62]</sup> has a Voronoi-based map generator in Unity, with [source code](#)<sup>[63]</sup>. He uses tectonic plates with uplift to generate the mountains.
- Xiang Wei has a [spherical world generator](#)<sup>[64]</sup>.
- Alexey Nabrosov wrote a map generator and wrote a [page about the process](#)<sup>[65]</sup> (Russian), and released [source code](#)<sup>[66]</sup>.
- Little Cube Valley<sup>[67]</sup> uses Voronoi maps with aquifers, wind, temperature, drainage, salinity to determine rivers and biomes.
- Luca Morselli's [map generator page](#)<sup>[68]</sup> has [interactive demos](#) of each step of the generation
- Nortantis<sup>[69]</sup> uses plate tectonic simulation to create islands and continents
- Nexus Maps<sup>[70]</sup> by nexus723 ([reddit](#)<sup>[71]</sup>) uses Voronoi with weather, coastlines, rivers, biomes, settlements, roads, trade routes, nations, ports, and other features.
- Vagabond<sup>[72]</sup> uses Voronoi to construct wilderness maps with rivers, towns, climates, roads, forests.

### 13.2. Similar algorithms in other programming languages

- Richard Janicek has a [Haxe version](#)<sup>[73]</sup> that compiles to Javascript+Canvas, with a [demo](#)<sup>[74]</sup> {broken as of 2023}, and also a [Javascript version](#)<sup>[75]</sup>, with a demo.
- Jeff Terrace has contributed patches and also wrote [a utility to convert the XML into COLLADA](#)<sup>[76]</sup>, as well as [a COLLADA viewer](#)<sup>[77]</sup> in WebGL.
- Alex Schröder has been working on a [Perl version](#)<sup>[78]</sup> that generates SVG output. {screenshots broken as of 2023}
- Christopher Garrett has written a [Voronoi/Delaunay port for iOS](#)<sup>[79]</sup>.
- Adam Martin maintains a [Voronoi/Delaunay port for Unity](#)<sup>[80]</sup>, forked from [Julian Ceipek's port](#)<sup>[81]</sup>.
- Christophe Guebert has written a [Java+Processing version](#)<sup>[82]</sup>.
- Baran Kahyaoglu has written a [C#/NET version](#)<sup>[83]</sup>, and abhimir has a [Unity version](#)<sup>[84]</sup> of [Chris Herborth](#)<sup>[85]</sup>'s version of Baran's code.
- Tommy Waters has [another Javascript version](#)<sup>[86]</sup>, with [source](#)<sup>[87]</sup> and a [blog post](#)<sup>[88]</sup>.
- Erop Xapbar has a [Haxe/NME version](#)<sup>[89]</sup>.
- Connor Clark has a [Java version](#)<sup>[90]</sup>.
- Kylepixel has a [Javascript project](#)<sup>[91]</sup> that also uses Voronoi regions for map generation.
- Nuclear Horse Studios has a [C#/Unity version](#)<sup>[92]</sup> (MIT license).

- Stafford Williams also has a [C#/Unity version](#)<sup>[93]</sup> (MIT license).
- Gareth Higgins has a [C# map generator](#)<sup>[94]</sup>.
- Martín Candela Calabuig has a [C++ version](#)<sup>[95]</sup>.
- Tobias is working on a [C++ version](#)<sup>[96]</sup>.
- Spencer Judge has [C++ code](#)<sup>[97]</sup> for a map generator using these techniques.
- BitAlchemists has a [Dart version](#)<sup>[98]</sup> and [source](#)<sup>[99]</sup> (MIT license).
- Thomas R. Koll has a [Lua port](#)<sup>[100]</sup> that he's using for [Autonomous Planetary Research Individual 50](#)<sup>[101]</sup>.
- [Island](#)<sup>[102]</sup> is a Scala project that uses techniques similar to the ones on this page.
- [PtolemyJS](#)<sup>[103]</sup> is a Javascript game engine that includes a Voronoi region map generator.
- Jay Stevens has a [C++ Unreal 4 version](#)<sup>[104]</sup> (MIT license).
- Carl Wieland has a [Swift version](#)<sup>[105]</sup>, no license given.
- Steve Johnstone has a [C#/Unity version](#)<sup>[106]</sup> (MIT license).
- DeiveEx has a [C#/Unity version](#)<sup>[107]</sup>, no license given.

### 13.3. Other projects

- [PolyWorld](#)<sup>[108]</sup> is based on Connor Clark's Java code, and is now part of the Terasology game (watch [this video](#)<sup>[109]</sup>).
- [Besiege](#)<sup>[110]</sup> uses these techniques for their procedural map generator.
- [TerraFirmaCraft 2](#)<sup>[111]</sup> uses these techniques for their game, with a hex grid.
- [Vagabond](#)<sup>[112]</sup> uses these techniques for their world generator.

The map generator wasn't designed for direct use but Welsh Piper ([encounter tables](#)<sup>[113]</sup>, [Minocra](#)<sup>[114]</sup>, [Kingdoms in Trevail](#)<sup>[115]</sup>, [Cresta](#)<sup>[116]</sup>, [Life & Thorn](#)<sup>[117]</sup> (comics), and others ([1](#)<sup>[118]</sup>, [2](#)<sup>[119]</sup>, [3](#)<sup>[120]</sup>, [4](#)<sup>[121]</sup>, [5](#)<sup>[122]</sup>, [6](#)<sup>[123]</sup>, [7](#)<sup>[124]</sup>, [8](#)<sup>[125]</sup>, [9](#)<sup>[126]</sup>, [10](#)<sup>[127]</sup>, [11](#)<sup>[128]</sup>, [12](#)<sup>[129]</sup>, [13](#)<sup>[130]</sup>), are using the map generator to create maps for their projects. You can use the "export PNG" button to export a 2048x2048 PNG that you can then adapt with Photoshop to color, style, and annotate for your own needs.

### 14.1. Modules

I tried to structure the map representation so that modules could annotate them without creating a code dependency. The GUI module `mapgen2.as` depends on `Map.as` (core) and `Roads.as` (non-core), but `Maps.as` does *not* depend on `Roads.as`. Every polygon, edge, and corner in the graph has an index, which can be used as a key into an external table. In `Roads.as`, there's a `road` array that's indexed by the edge index.

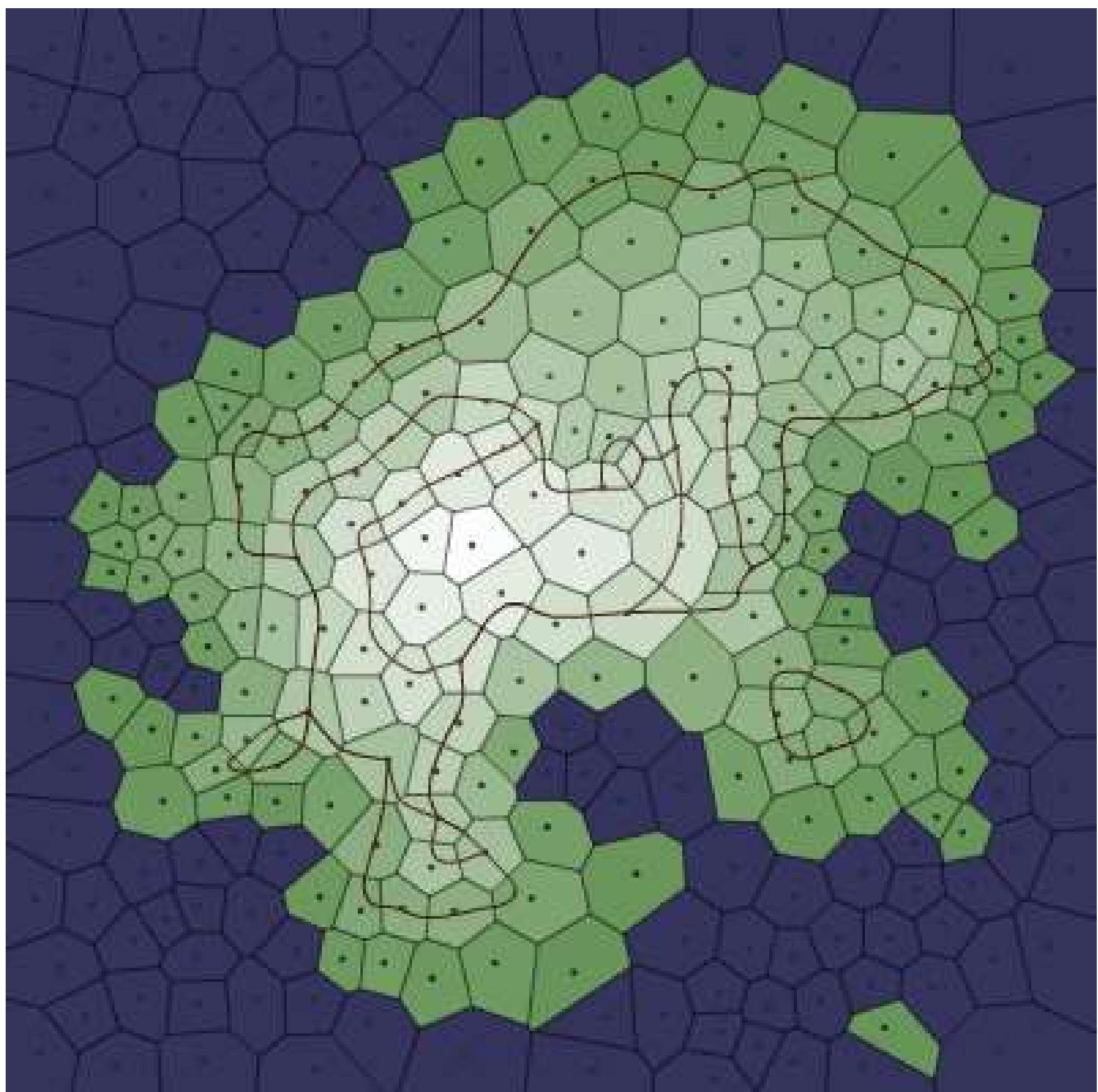
Where core map code can reference `edge.river` as a core field, the module can't do that. Instead, the module references its local variable `road[edge.index]`. This works for polygon centers and corners as well. It keeps the core clean.

I have three modules: Roads, Lava, and NoisyEdges.

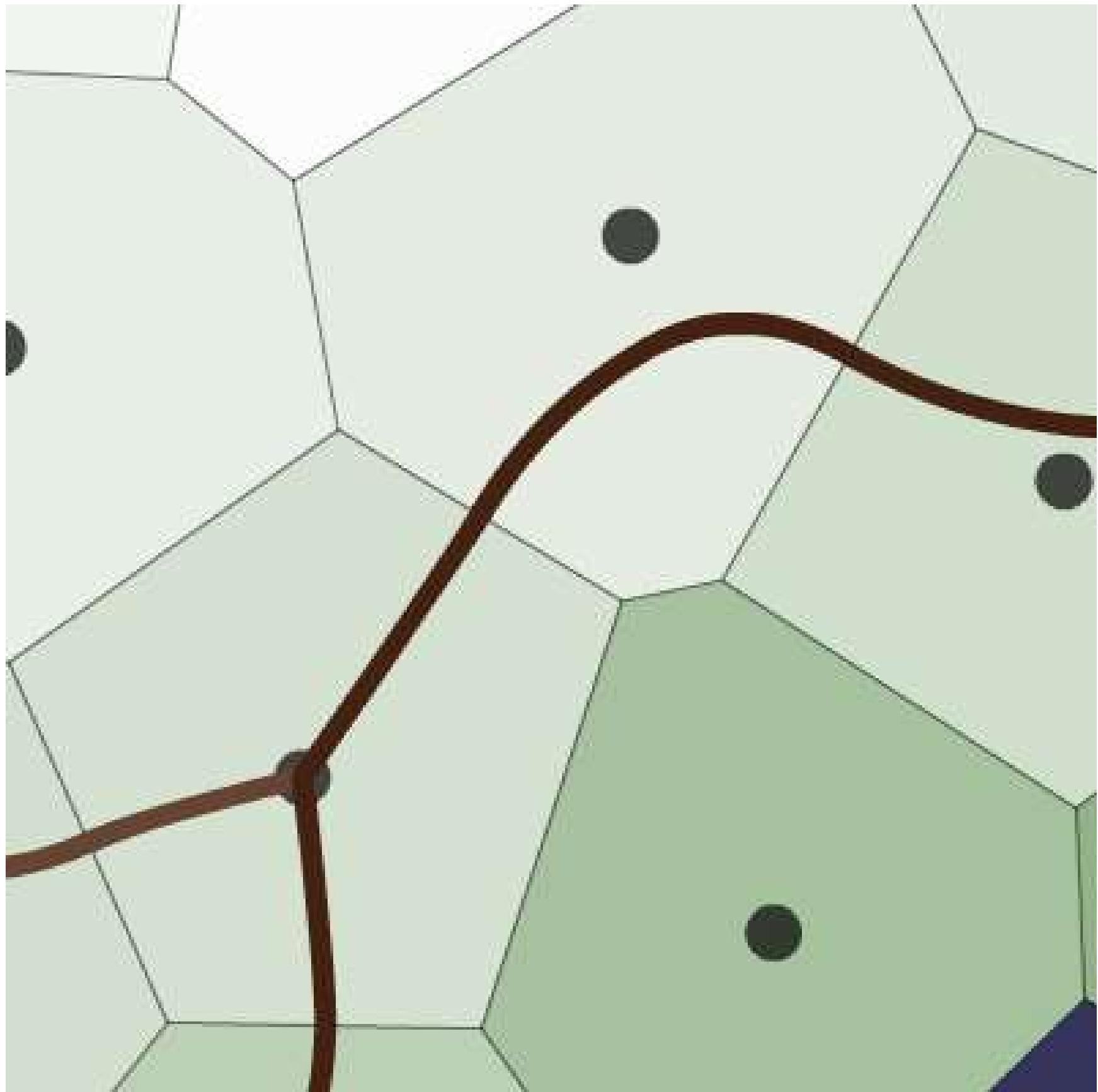
### 14.2. Roads

Realm of the Mad God doesn't use most of the features of this map generator, but I built a road generator for them. I observed that in the game, people naturally explore rivers. This unfortunately leads them up to the mountains, where they die. I wanted to build some roads that are at right angles to the rivers.

I calculated contour lines along the corners. Where the contour level changes, there's a road. It's a fairly simple algorithm that works most of the time, but sometimes creates tiny loops:



Whereas rivers meander along Voronoi edges (blue lines in the diagram above), roads go on Delaunay edges (red lines). Roads don't get the noisy edge treatment. Instead, they are drawn with splines from edge midpoint to midpoint:

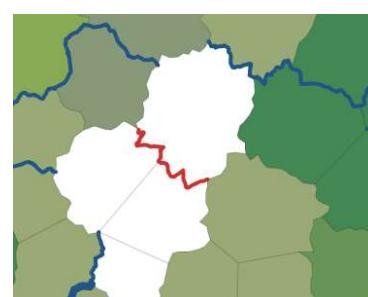


Most polygons have two neighbors with roads. For them, there's a regular spline connecting the two edge midpoints. For polygons that have more than two road neighbors, I instead draw an intersection, with splines from all the edge midpoints to the polygon center. In the diagram above, the lower left polygon has an intersection and the upper right polygon has a regular spline.

Roads cross the edges while rivers follow the edges, so that makes it easy to turn a road into a bridge when it crosses the river.

#### 14.3. Lava

Lava and rivers follow the same paths. Lava fissures occur in high dry areas, and are assigned to some subset of the edges. In a game, lava and water will of course be different, but here they only differ in color and placement. Lava gets the noisy edge treatment:



#### 15.1. Abstract Rendering

A map should show the relevant portions of the world, not all the details. With this project I'm generating maps with some level of detail, but it's not detailed down to the vegetation or towns, and it's not completely abstract either. It may be possible to render a more abstract map in the style of maps of Middle Earth (such as [this one](#)<sup>[131]</sup>). I made some notes about [what I want from game maps](#)<sup>[132]</sup>.

#### 15.2. Watersheds

By following the downslope arrows (described in the section on elevation), there's a path from every polygon corner, along edges, to the coast. I use this to mark every corner with the location where the water would flow out to the ocean. All corners with the same outflow location can be considered to be part of the same watershed.

The watershed code is incomplete. There's a watershed view in the demo, but I'm not happy with it. I've tried polygon centers and corners as watershed boundaries and neither is quite right. I have put watershed calculations on hold until the day I need them.

The place I thought watersheds would be useful is for naming [larger regions](#)<sup>[133]</sup>. There are roughly 1000 land polygons on the map in the demo. In a game map it might be nice to have a smaller number of named regions that group together polygons. For example, the XYZ Mountains can be above the XYZ Valley, which might have the XYZ River flowing through it. Players would be able to learn that these features are related. I didn't get very far with this project but I might come back to it someday.

### 15.3. Impassable borders

In this map generator all the borders between polygons are the same. There's a smooth transition from one to the other. It might be interesting to make some edges discontinuous, so that we can make cliffs, chasms, plateaus, and other sudden elevation changes. See [this article](#)<sup>[134]</sup> for ideas on how to make the Voronoi regions interesting for gameplay.

### 15.4. Terrain analysis

Pathfinding should be fairly fast on a polygonal map, and may be useful for terrain analysis. For example, if two locations are spatially close but pathwise far, that may mean there's a bay or mountain in the way, and that'd be a good place for a tunnel or bridge. A pathfinder could also help find places where we need bridges to nearby islands. Polygons that show up on paths often might be strategically more valuable than polygons that rarely are used for paths.

### 15.5. Named areas

As mentioned in the watersheds section, I'd like to name map features. Combined with terrain analysis, names could be assigned to rivers, mountains, lakes, groups of polygons, coastlines, oceans, forests, peninsulas, valleys, etc. Names in an area could be related. For example, the XYZ River could flow from Mount XYZ through the XYZ Valley. I haven't worked on this in part because I think it helps if it's for a specific game instead of a generic map generator. Not only should names connect to the game's theme, there could be items and quests and plot elements that are related. For example, the Sword of XYZ might be found only in the XYZ Valley.

### 15.6. Variable density

Fortune's algorithm should work within a polygon to subdivide it into smaller polygons. A map where most of the world is coarse, but some areas are more finely divided, could be interesting. Alternatively, we could place the original points with variable density so that some areas get more polygons than others. For example, we could use [Poisson Disk Sampling](#)<sup>[135]</sup> instead of Lloyd's Algorithm.

### 15.7. Better noisy edges

I implemented a very simple noisy edge system, with jagged lines. The corners are very visible when you zoom in on the map. A better system might involve curved splines, or a fractal expansion that looks more detailed as you zoom in.

16

## Appendix: Process improvements

#

This page started out as three blog posts on the topic: [part 1](#)<sup>[136]</sup> is about polygons, map representation, islands, oceans and lakes and beach and land; [part 2](#)<sup>[137]</sup> is about elevations, rivers, moisture, and biomes; and [part 3](#)<sup>[138]</sup> is about rendering, demo, and source code. Having it all on one page made more sense in the long term.

For those of you interested in how I got to this point, a brief history:

- In December 2009, Rob and Alex of Wild Shadow Studios had asked me if I had a quick way to generate maps. I had already been thinking about using Perlin/Simplex noise for maps, so I tried it, with [good results](#)<sup>[139]</sup>. I got something going within a day, and then spent the next month tweaking and trying variations. Most of the variations failed, and taught me that there were limitations with this approach. One month was a good time to stop, so I stopped working on maps and moved on to other small projects – art, animation, monster groups, NPC AI, among others.
- In June 2010, I was inspired to work on maps again. I spent the month sketching out ideas on paper and trying some prototypes. I tried hexagon grids, hexagonal river basins, quadrilateral river generation, volcanos, hills, erosion, weather systems, and a few other things. **Everything failed**. However, I learned a lot by trying these things out. Delaunay triangulations for example didn't work out, but they led me to Voronoi diagrams. The quadrilateral river generation didn't work out, but the quadrilaterals were useful later when I worked on noisy edges. The erosion system didn't work out, but some of the same ideas were useful when I worked on rivers.
- While attending the Procedural Content Generation workshop, I sketched out some more ideas on paper for map generation. During the July 4 long weekend, I implemented these, and they worked really well. I worked out Voronoi polygons, map representation, island generation, noisy edges, elevation, biomes, rivers, and elevation redistribution that weekend. I experienced [flow](#)<sup>[140]</sup>. It was great! Most of the core system was in place in only three days.
- Every weekend in July and August, I made improvements, many of them substantial. I also made many changes that didn't work out, and I deleted them. As the core map features became good, I shifted my focus to the map rendering and the UI. As the map rendering and UI improved, I was able to see more flaws in the maps, and I found lots more bugs I needed to fix. I also made major refactorings to simplify the code that had grown organically.
- By the end of August I found that I was only working on minor things, and decided the project was ready to wrap up. I spent the Labor Day long weekend writing up the results on this page (and the blog posts). Much of my time went into making good diagrams. The diagrams actually exposed more bugs, and I ended up making bug fixes, greatly simplifying one feature (elevation redistribution), and implementing a new feature (moisture redistribution). I also renamed and commented code to make it easier to explain.

Why am I keeping track of this? It's because I'm trying to improve the process by which I approach these small (1-3 month) projects. There are some things I want to remember:

1. It's useful to have a **key idea** that drives everything else. The simple maps I did in January were based on Perlin Noise. These maps are based on Voronoi diagrams. I need to pick something and go with it, but only after...
2. It sometimes takes a **lot of experimentation** before I run across the right idea. I spent a month on ideas before coming up with Voronoi as the key structure. I need to sketch out lots and lots of ideas.
3. I have a **lot of failures**. The key is to fail quickly, not to avoid failing. I need to not get discouraged.
4. I got the core system up in three days. A **quick prototype** can tell me a lot about whether an idea's going to work out. In the early stages I need to focus on a prototype and not worry about making it a high quality system.
5. In the very early phase it's more important to **learn from the system** than to produce good code. I need to ask myself what I'm trying to learn with a prototype.
6. **Failures are sometimes useful later**. I need to keep them accessible. I've been deleting the code as soon as it fails, but maybe I should make lots more git branches and store them there.
7. **The ability to see things** can help a great deal in understanding what's going on. I missed several bugs because I never bothered to build a visualization for that part of the data. I need to display as much as I can.
8. There are sometimes **tiny things that seem wrong** that actually mean I have a bug somewhere. I often shrug these things off. Even if it's not a good time to investigate and fix some bug, I need to track them somewhere so that I can later investigate.
9. **Writing the blog posts** helped tremendously. It forced me to understand every part of the system, to look at all the data, and to make sure all the code could be understood. It made me question every phase of map generation and improve the ones that were hard to explain. I need to write blog posts much earlier in the process. Explaining is a good way to learn.

I'll keep these in mind as I work on future projects.

17

## Appendix: References

#

Thanks to the [Dungeon League blog](#)<sup>[141]</sup> for a great series on procedural map generation, the [Procedural Content Generation wiki](#)<sup>[142]</sup> for [ideas for map generation](#)<sup>[143]</sup>, the incomplete [Voronoi wiki](#)<sup>[144]</sup> for some useful resources about Voronoi diagrams.

My thanks to [nodename](#) for [as3delaunay](#)<sup>[145]</sup>. It's an Actionscript 3 library for generating Voronoi and Delaunay graphs. Also my thanks to [polygonal](#) for the [PM\\_PRNG](#)<sup>[146]</sup> random number library, which allows me to use and reset the seed value so that I can reproduce a series of pseudorandom numbers. I used the [OptiPNG library](#)<sup>[147]</sup> to optimize the PNG images on this page.

[Delaunay triangulation](#)<sup>[148]</sup> and [Voronoi polygons](#)<sup>[149]</sup> are taught in many graphics classes. For example, see [Stanford CS 368](#)<sup>[150]</sup>(PDF). I also looked at [relative neighborhood graphs](#)<sup>[151]</sup> and [Gabriel graphs](#)<sup>[152]</sup> but didn't use either.

Delaunay triangulation can be used for maps, in the form of [triangular irregular networks](#)<sup>[153]</sup>. Voronoi diagrams are also used for maps. For example, see [this blog post about using Voronoi for Unhangband biomes](#)<sup>[154]</sup>.

[Fortune's Algorithm](#)<sup>[155]</sup> is one of several algorithms that can turn a set of points into Voronoi polygons. It's implemented in [as3delaunay](#). [Lloyd Relaxation](#)<sup>[156]</sup> is used to improve the distribution of random points. It decreases the irregularity of the Voronoi polygons.

The Voronoi wiki has some incomplete pages on [graph representation](#)<sup>[157]</sup> and [edge representation](#)<sup>[158]</sup>, as well two pages that helped me with river generation: [rivers and watersheds](#)<sup>[159]</sup> and [crust and skeleton](#)<sup>[160]</sup>.

The [Relief Shading Website](#)<sup>[161]</sup> has some [images of shaded relief maps](#)<sup>[162]</sup>, as well as design guidelines for shading and coloring. I am sad to say I did not have time to apply these techniques. I also studied Bing and Google maps to see how they draw various features; see [this article](#)<sup>[163]</sup> and [my blog post](#)<sup>[164]</sup> and [this article](#)<sup>[165]</sup>.

The [Whittaker diagram](#)<sup>[166]</sup> is one way to predict common biomes given climate. Wikipedia has a page listing [biome classification schemes and various biomes](#)<sup>[167]</sup>.

Wikipedia also has a nice [list of landforms](#)<sup>[168]</sup> that one might want to generate in a game map generator. I did not explore these for this project.

Ken Perlin created Perlin Noise, and then he made a sequel, [Simplex Noise](#)<sup>[169]</sup> [PDF]. For this project I used Perlin Noise for the overall island shape. I used Perlin instead of Simplex because Flash included it in the standard library, but I use Simplex for all my newer projects. Joe Slayton has a page on [how to turn perlin noise into islands](#)<sup>[170]</sup>. I was looking for "blue noise", which led me to [Dunbar and Humphreys's paper on noise generation](#)<sup>[171]</sup> and [recursive Wang tiles](#)<sup>[172]</sup> before I found Lloyd's algorithm for better random point distribution. I also looked at [Craig Reynold's textures](#)<sup>[173]</sup> but didn't have time to do anything with them.

Also interesting were [these papers about generating worlds](#)<sup>[174]</sup> (the author has continued writing interesting papers since I published this page in 2010), [Gozzy's wilderness map generator](#)<sup>[175]</sup>, [donjon's world generator](#)<sup>[176]</sup>, a [paper on procedural road generation](#)<sup>[177]</sup>. [Straight skeletons](#)<sup>[178]</sup> seemed like they might be useful for defining mountain ranges, but once I discovered how well "distance from coast" worked, I didn't need anything else. The 3d map generation in [Dwarf Fortress](#)<sup>[179]</sup> is pretty neat.

Email me [redblobgames@gmail.com](mailto:redblobgames@gmail.com), or tweet [@redblobgames](#), or comment:

### Endnotes

- [1]: <https://www.redblobgames.com/maps/mapgen2/>
- [2]: [https://en.wikipedia.org/wiki/Realm\\_of\\_the\\_Mad\\_God](https://en.wikipedia.org/wiki/Realm_of_the_Mad_God)
- [3]: [https://en.wikipedia.org/wiki/Voronoi\\_diagram](https://en.wikipedia.org/wiki/Voronoi_diagram)
- [4]: [http://www.voronoi.com/index.php?title=Voronoi\\_Applications](http://www.voronoi.com/index.php?title=Voronoi_Applications)
- [5]: [http://www.voronoi.com/index.php?title=VoroWiki\\_main\\_page](http://www.voronoi.com/index.php?title=VoroWiki_main_page)
- [6]: <https://nodename.github.io/as3delaunay/>
- [7]: [https://en.wikipedia.org/wiki/Fortune's\\_algorithm](https://en.wikipedia.org/wiki/Fortune's_algorithm)
- [8]: <http://bit-player.org/2011/a-slight-discrepancy>
- [9]: [https://en.wikipedia.org/wiki/Lloyd's\\_algorithm](https://en.wikipedia.org/wiki/Lloyd's_algorithm)
- [10]: [https://en.wikipedia.org/wiki/Centroid#Centroid\\_of\\_polygon](https://en.wikipedia.org/wiki/Centroid#Centroid_of_polygon)
- [11]: <https://www.redblobgames.com/x/1721-voronoi-alternative/>



[124]: [https://old.reddit.com/r/worldbuilding/comments/5wme1e/made\\_a\\_map\\_of\\_my\\_world\\_sphaira\\_with/](https://old.reddit.com/r/worldbuilding/comments/5wme1e/made_a_map_of_my_world_sphaira_with/)  
[125]: <https://togetter.com/li/866956>  
[126]: <https://twitter.com/halforddruid/status/978340419607052289>  
[127]: <https://twitter.com/fenhare/status/1024037692873437184>  
[128]: <https://vos.tumblr.com/post/162415550753/corianwornen-itsbio-nightshadeis>  
[129]: <https://davidvs.blogspot.com/2018/11/island-maps.html>  
[130]: <https://twitter.com/GhostArcher3/status/1212455077786669056>  
[131]: <http://geektyrant.com/news/2008/10/7/guillermo-del-toro-talks-about-the-entire-pentology-of-middl.html>  
[132]: <http://www-cs-students.stanford.edu/~amitp/game-programming/skyrim-maps/>  
[133]: [https://en.wikipedia.org/wiki/Drainage\\_basin](https://en.wikipedia.org/wiki/Drainage_basin)  
[134]: <http://roguelikedeveloper.blogspot.com/2007/07/wilderness-generation-using-voronoi.html>  
[135]: <http://devmag.org.za/2009/05/03/poisson-disk-sampling/>  
[136]: <http://simblob.blogspot.com/2010/09/polygon-map-generation-part-1.html>  
[137]: <http://simblob.blogspot.com/2010/09/polygon-map-generation-part-2.html>  
[138]: <http://simblob.blogspot.com/2010/09/polygon-map-generation-part-3.html>  
[139]: <https://simblob.blogspot.com/2010/01/simple-map-generation.html>  
[140]: [https://en.wikipedia.org/wiki/Flow\\_\(psychology\)](https://en.wikipedia.org/wiki/Flow_(psychology))  
[141]: <http://www.dungeoneleague.com/>  
[142]: <http://pcg.wikidot.com/>  
[143]: <http://pcg.wikidot.com/pcg-algorithm:map-generation>  
[144]: [http://www.voronoi.com/wiki/index.php?title=VoroWiki\\_main\\_page](http://www.voronoi.com/wiki/index.php?title=VoroWiki_main_page)  
[145]: <http://nodename.github.io/as3 delaunay/>  
[146]: <http://lab.polygonal.de/2007/04/21/a-good-pseudo-random-number-generator-prng/>  
[147]: <https://optipng.sourceforge.net/>  
[148]: [https://en.wikipedia.org/wiki/Delaunay\\_triangulation](https://en.wikipedia.org/wiki/Delaunay_triangulation)  
[149]: [https://en.wikipedia.org/wiki/Voronoi\\_diagram](https://en.wikipedia.org/wiki/Voronoi_diagram)  
[150]: [http://graphics.stanford.edu/courses/cs368-06-spring/handouts/Delaunay\\_1.pdf](http://graphics.stanford.edu/courses/cs368-06-spring/handouts/Delaunay_1.pdf)  
[151]: [https://en.wikipedia.org/wiki/Relative\\_neighborhood\\_graph](https://en.wikipedia.org/wiki/Relative_neighborhood_graph)  
[152]: [https://en.wikipedia.org/wiki/Gabriel\\_graph](https://en.wikipedia.org/wiki/Gabriel_graph)  
[153]: [https://en.wikipedia.org/wiki/Triangulated\\_irregular\\_network](https://en.wikipedia.org/wiki/Triangulated_irregular_network)  
[154]: <http://roguelikedeveloper.blogspot.com/2007/07/wilderness-generation-using-voronoi.html>  
[155]: [https://en.wikipedia.org/wiki/Fortune's\\_algorithm](https://en.wikipedia.org/wiki/Fortune's_algorithm)  
[156]: [https://en.wikipedia.org/wiki/Lloyd's\\_algorithm](https://en.wikipedia.org/wiki/Lloyd's_algorithm)  
[157]: [http://www.voronoi.com/wiki/index.php?title=PAN\\_graphs](http://www.voronoi.com/wiki/index.php?title=PAN_graphs)  
[158]: [http://www.voronoi.com/wiki/index.php?title=Spatial\\_Structures](http://www.voronoi.com/wiki/index.php?title=Spatial_Structures)  
[159]: [http://www.voronoi.com/wiki/index.php?title=Rivers\\_and\\_watersheds](http://www.voronoi.com/wiki/index.php?title=Rivers_and_watersheds)  
[160]: [http://www.voronoi.com/wiki/index.php?title=Crust\\_and\\_Skeleton](http://www.voronoi.com/wiki/index.php?title=Crust_and_Skeleton)  
[161]: <http://www.relfshading.com/>  
[162]: <http://www.relfshading.com/examples/>  
[163]: <http://web.archive.org/web/20101223024222/http://www.41latitude.com/post/897973389/bing-maps-redesign>  
[164]: <http://amitp.blogspot.com/2007/09/online-map-sites.html>  
[165]: [http://infosthetics.com/archives/2010/06/bing\\_destination\\_map\\_automatic\\_napkin\\_sketching\\_of\\_maps.html](http://infosthetics.com/archives/2010/06/bing_destination_map_automatic_napkin_sketching_of_maps.html)  
[166]: <http://pcg.wikidot.com/pcg-algorithm:whittaker-diagram>  
[167]: <https://en.wikipedia.org/wiki/Biome>  
[168]: [https://en.wikipedia.org/wiki/List\\_of\\_landforms](https://en.wikipedia.org/wiki/List_of_landforms)  
[169]: <https://github.com/stegu/perlin-noise/blob/master/simplexnoise.pdf>  
[170]: <https://web.archive.org/web/20140718034013/http://www.ridgenet.net/~jslayton/FunWithWilburVol6/>  
[171]: <https://web.archive.org/web/20060410123742/https://www.cs.virginia.edu/~gfx/pubs/antimony/>  
[172]: [http://johanneskopf.de/publications/blue\\_noise/](http://johanneskopf.de/publications/blue_noise/)  
[173]: <http://www.red3d.com/cwr/texsyn/diary.html>  
[174]: <https://perso.liris.cnrs.fr/eric.galin/articles.html>  
[175]: <https://www.gozzys.com/wilderness-maps>  
[176]: <https://donjon.bin.sh/world/>  
[177]: <http://gram.cs.mcgill.ca/papers/rudzicz-08-iterated.pdf>  
[178]: [https://en.wikipedia.org/wiki/Straight\\_skeleton](https://en.wikipedia.org/wiki/Straight_skeleton)  
[179]: <http://www.bay12games.com/dwarves/>