Polygonal Map Generation for Games

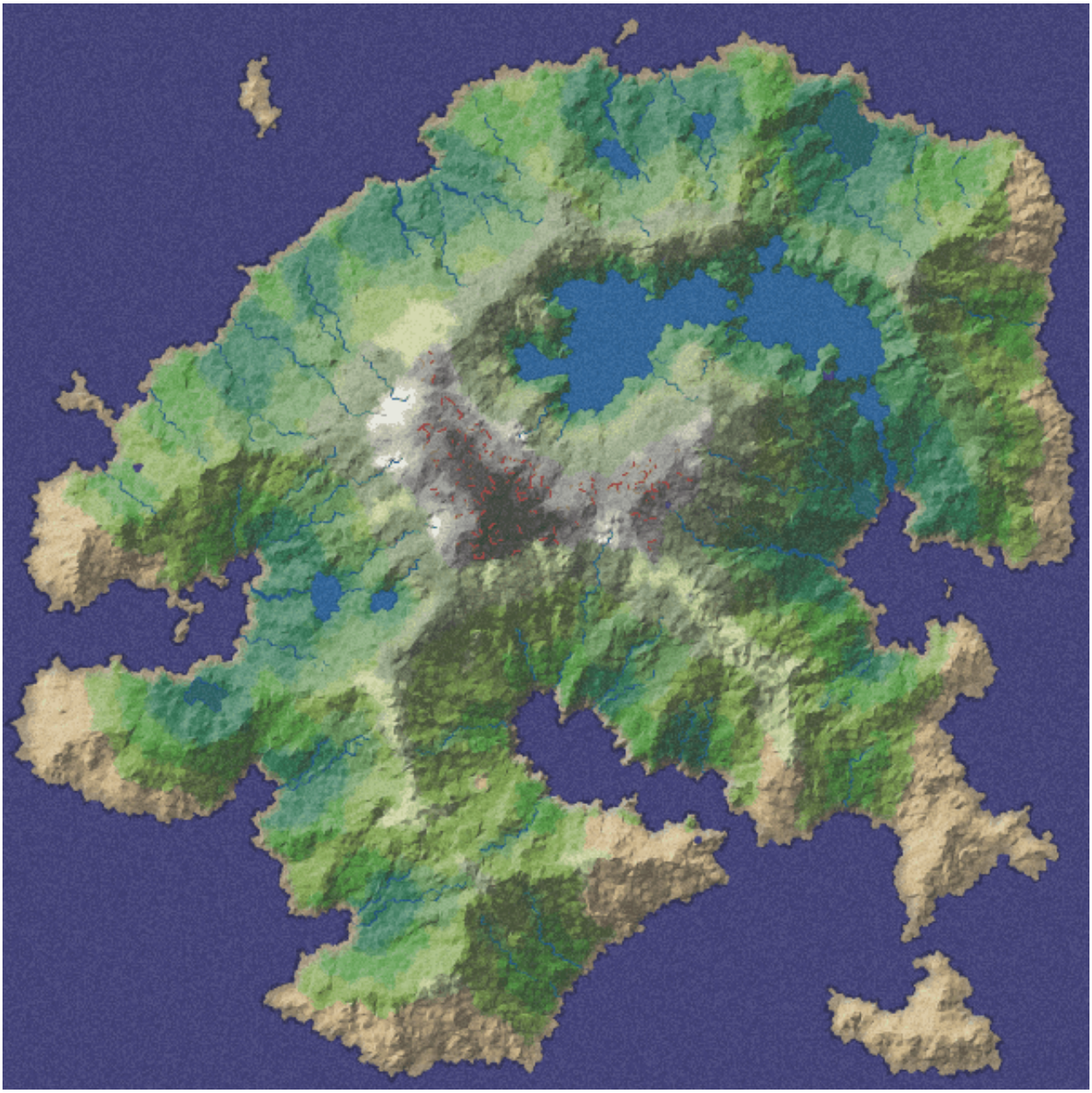
from Red Blob Games

4 Sep 2010

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 this time I decided to make polygonal maps. Instead of 1,000,000 tiles, what could I do with 1,000 polygons? I think the distinct player-recognizable areas might be useful for gameplay: locations of towns, places to quest, territory to conquer or settle, pathfinding waypoints, difficulty zones, etc. Once I have the polygons I can rasterize it back to tiles (or voxels), but I wanted to first generate the structure.

There were three main things I wanted: good coastlines, mountains and rivers. For the coastline, I wanted to make island/continent 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. For the rivers, I started with something simple: draw rivers from the coast to the mountains.

First, **try the demo!** Read on to learn how it works, or get the source code. Here's an example of the kinds of maps I want to produce:

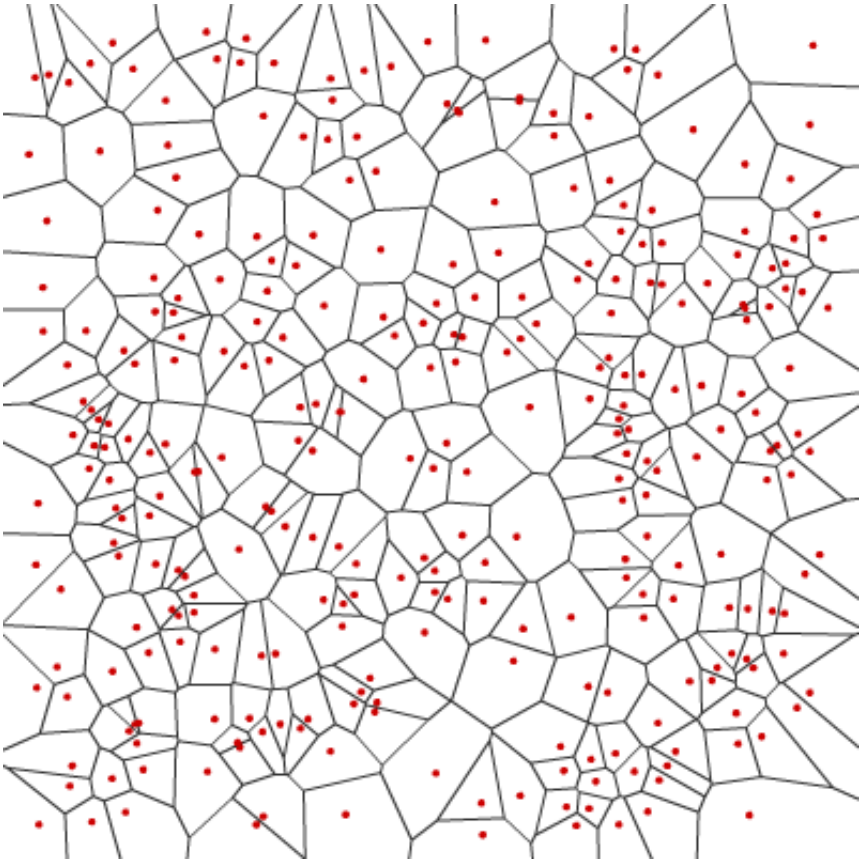


This page is a longer version of three blog posts on the topic: part 1 is about polygons, map representation, islands, oceans and lakes and beach and land; part 2 is about elevations, rivers, moisture, and biomes; and part 3 is about rendering, demo, and source code.

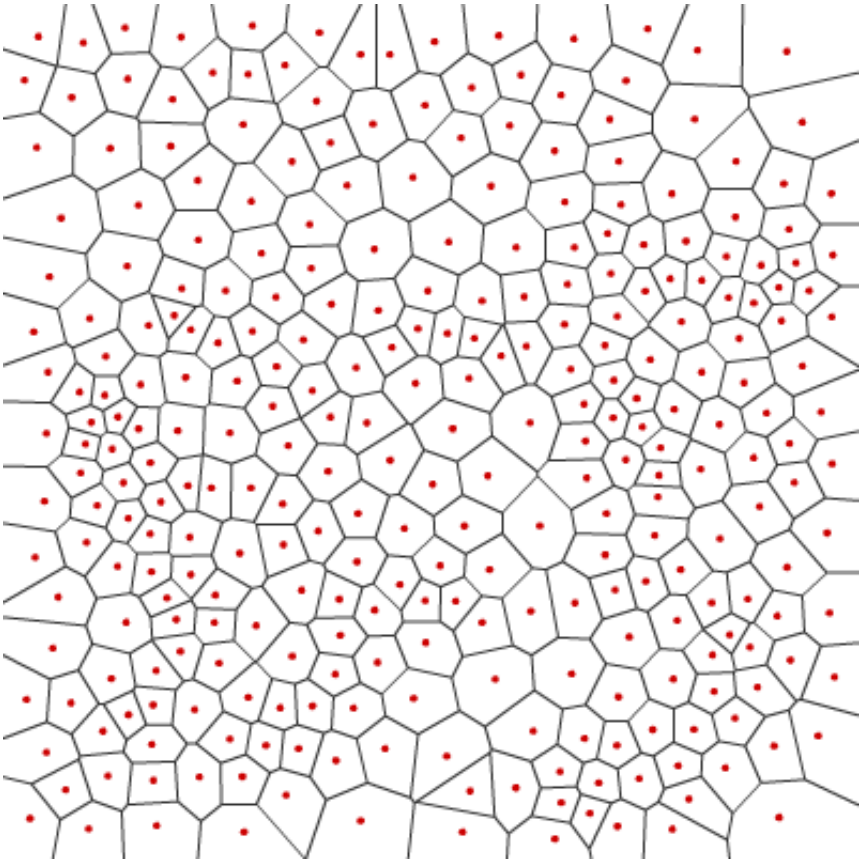
Polygons

The first step is to generate some polygons. I picked random points and generated Voronoi polygons, which are used for lots of things, including maps. The Voronoi wiki is incomplete but has some useful background. I'm using nodename's as3delaunay library, which has an implementation of Fortune's Algorithm.

Here's an example of random dots (red) and the polygons that result:



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, not random points. I approximate that by using a variant of Lloyd relaxation, 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 of the polygon. In my code I merely average the corners (see `improveRandomPoints`). Here's the result after running approximate Lloyd relaxation twice:



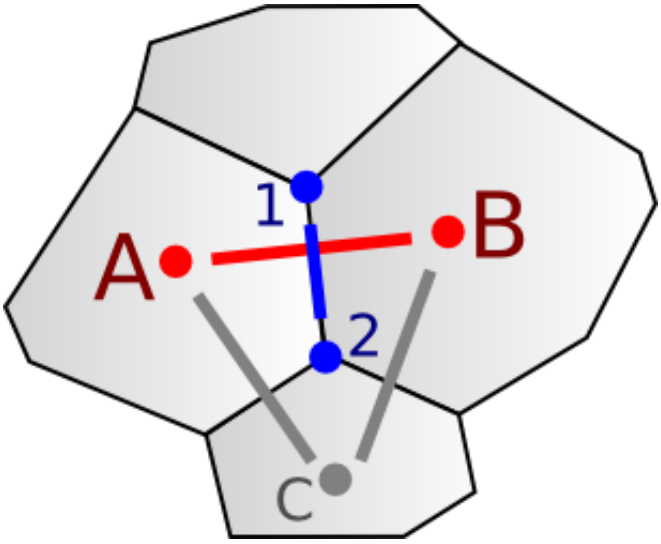
Compare it to running once or fifty times. The more iterations, the more regular the polygons get. Running it twice gives me good results but every game will vary in its needs.

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 will lose the Voronoi diagram properties. Those properties aren't used in this map generator, but keep this in mind if you want to use those properties in a game. You can either get better edge lengths or you can preserve the Voronoi distance properties.

Map Representation

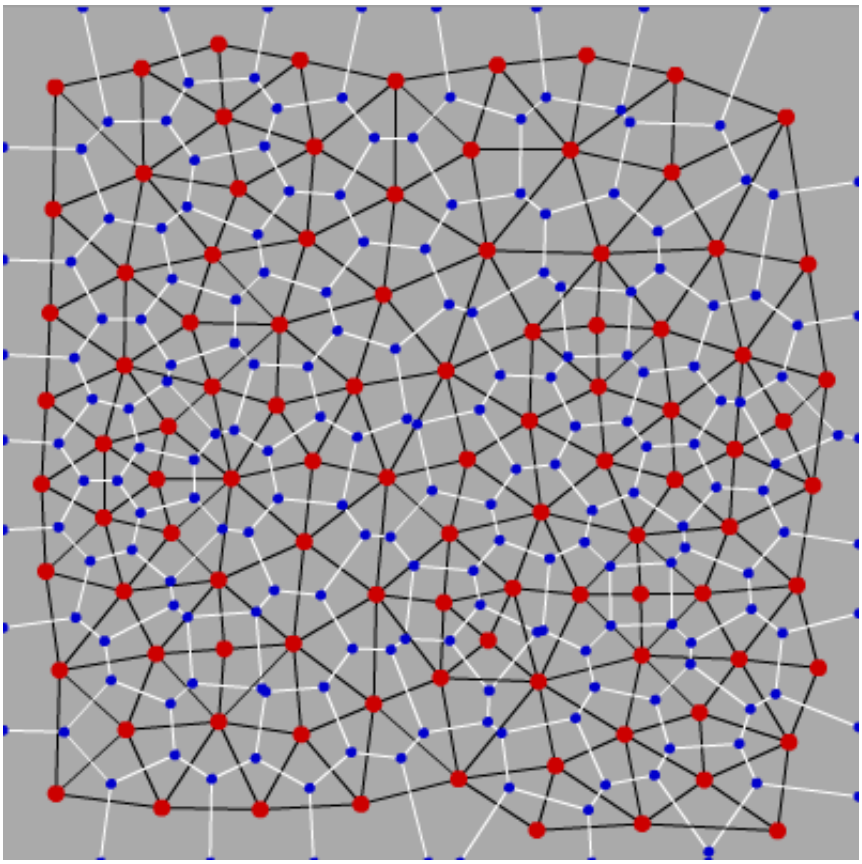
I'm representing the map as two related graphs: nodes and edges. The first graph has nodes for each polygon and edges between adjacent polygons. It represents the Delaunay triangulation, which is useful for anything involving adjacency (such as pathfinding). The second graph has nodes for each polygon *corner* and edges between corners. It contains the shapes of the Voronoi 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 for combining the data from the two graphs. In particular, edges can be shared. 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.

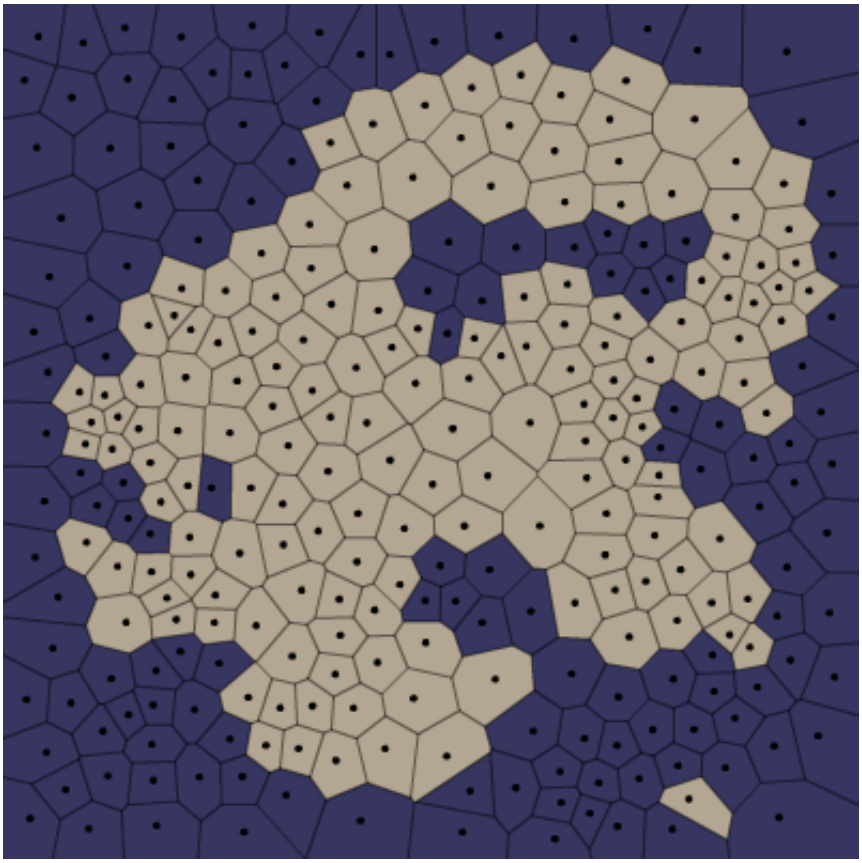
With the combined representation, I can now use the Relationships Between Grid Parts sections of my article on grids. They're not grids so I'm not assigning grid coordinates, but many of the algorithms that work on grids also work here, and the algorithms that work on graphs also work here (on either 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

Islands

The second step is to draw the coastline. I used a simple function to divide the world into land and water. There are many different ways to do this. You can even provide your own shapes, e.g., a skull island, although that code is not included in the demo. The map generator works with any division of points, but it forces the outer layer of polygons to be ocean. Here's an example that divides the world into land and water:



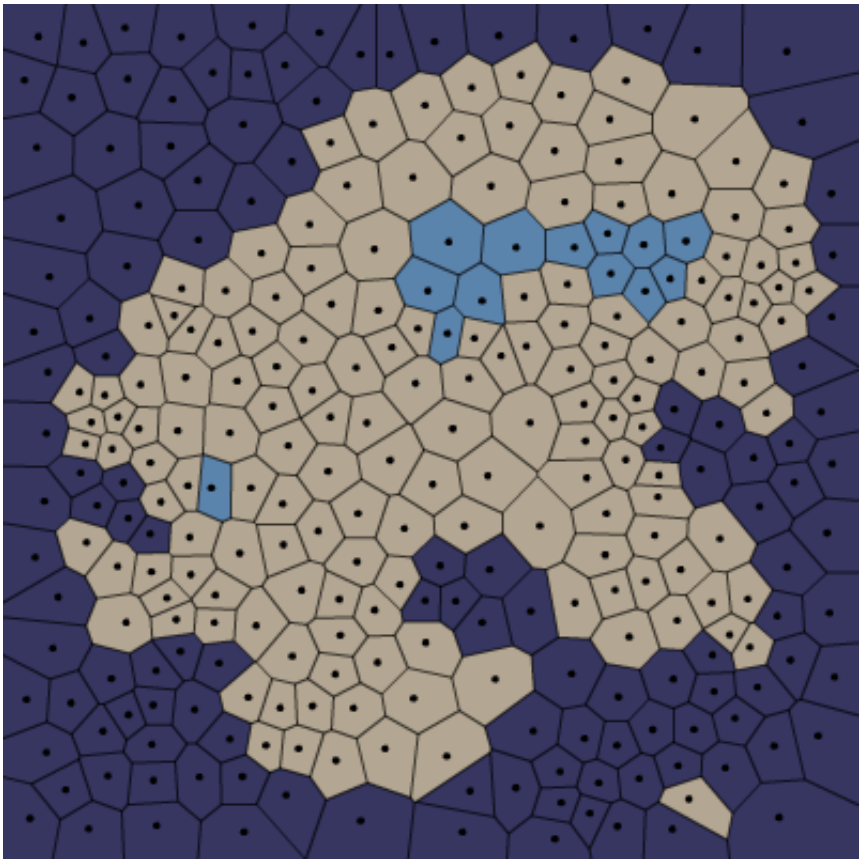
In the code, `Map.as` contains the core map generation code. The `IslandFunction` returns `True` if a position is land, and `False` for water. There are four island functions included in the demo:

- `Radial` uses sine waves to produce a round island
- `Perlin` uses Perlin noise to control the shape
- `Square` fills the entire map with land
- `Blob` draws my blob logo

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:

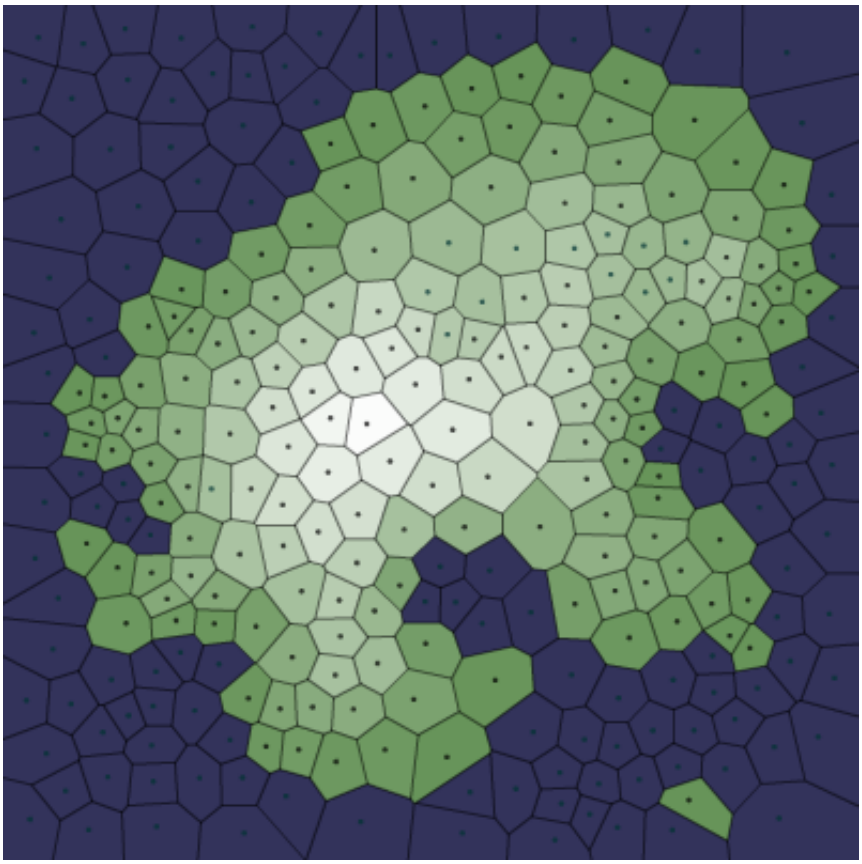
1. Set `Center.ocean` for any polygon connected to the borders of the map through water polygons. If `Center.water` is set but `.ocean` is not, then it's a lake.
2. Set `Center.coast` if the polygon is land but has an ocean border. Coastal areas will later get drawn as beaches.
3. Set `Corner.ocean` 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.

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`.

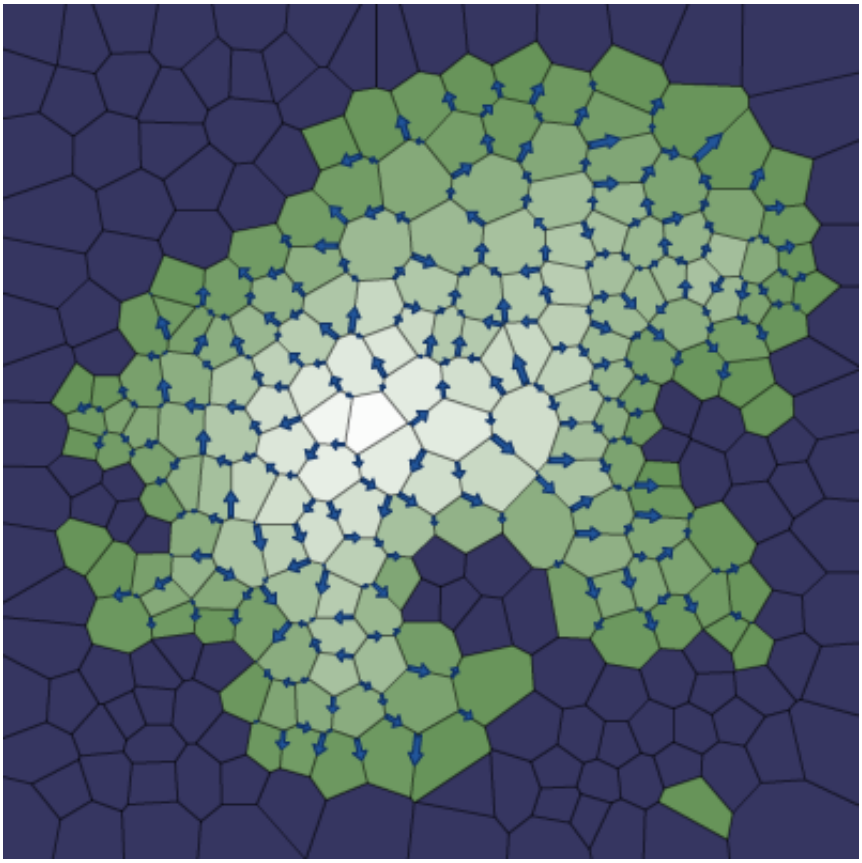
Water polygons don't count towards the distance. This is both because I expect lakes to be flat

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 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.

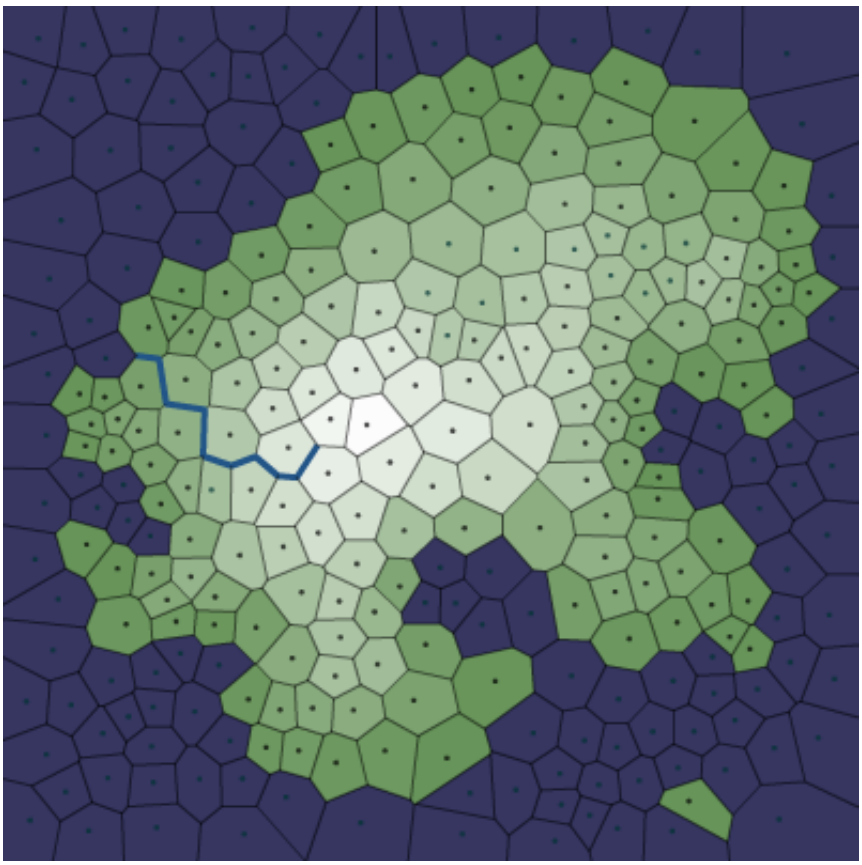
In addition, games may define their own use of elevation data. For example, Realm of the Mad God uses elevation to distribute monsters.

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.

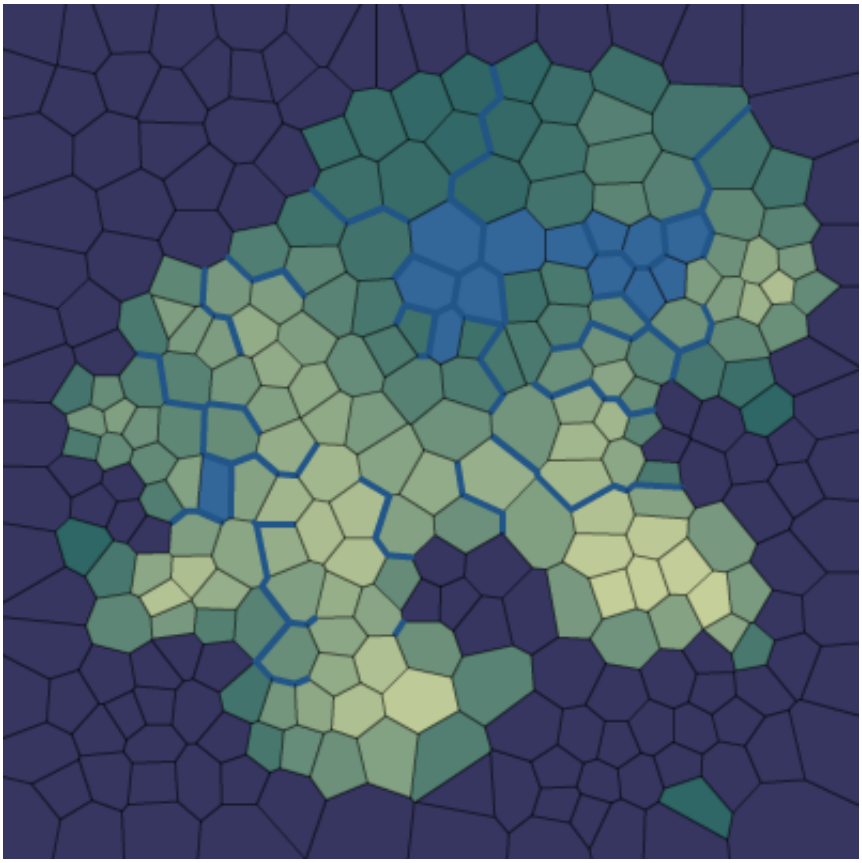
Rivers use the downhill directions shown earlier. I choose random corner locations in the mountains, and then follow the `Corner.downslope` path down to the ocean. The rivers flow from corner to corner:



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.

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 uses moisture and elevation to distribute vegetation.

Biomes

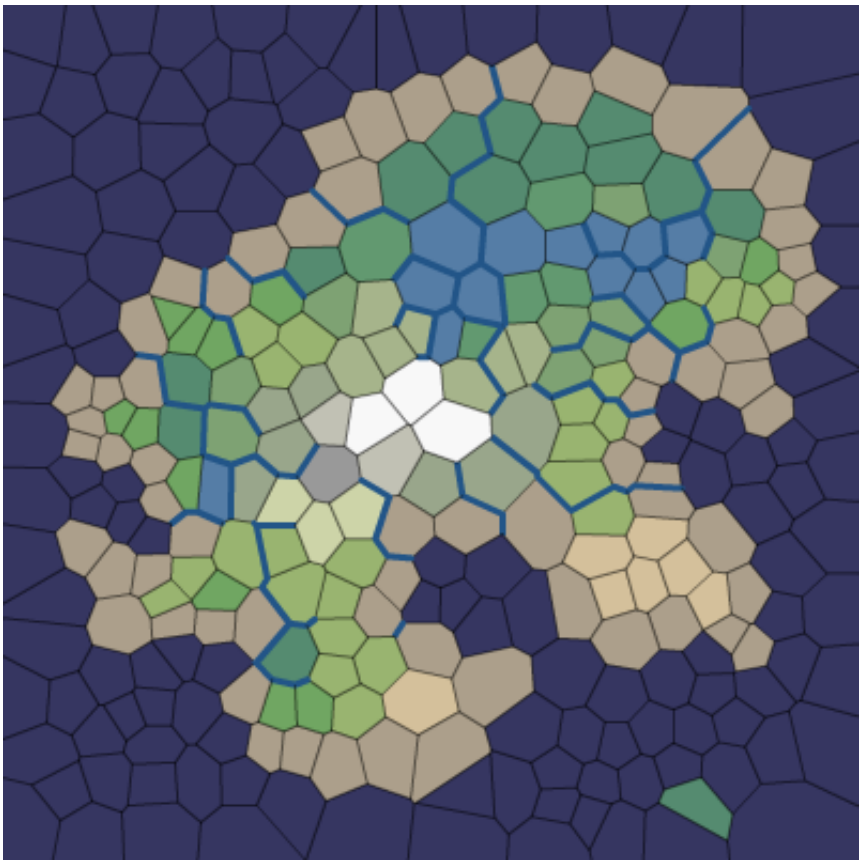
Together, elevation and moisture provide a good amount of variety to define biome types. I use elevation as a proxy for temperature. If this were a continent generator, latitude might be a contributor to temperature. Also, wind, evaporation, and rain shadows might be useful for transporting moisture 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 and adapted it to my needs:

Elevation Zone	Moisture Zone					
	6 (wet)	5	4	3	2	1 (dry)
4 (high)	SNOW			TUNDRA	BARE	SCORCHED
3	TAIGA		SHRUBLAND		TEMPERATE DESERT	
2	TEMPERATE RAIN FOREST	TEMPERATE DECIDUOUS FOREST		GRASSLAND		TEMPERATE DESERT
1 (low)	TROPICAL RAIN FOREST		TROPICAL SEASONAL FOREST		GRASSLAND	SUBTROPICAL DESERT

Here's the result:



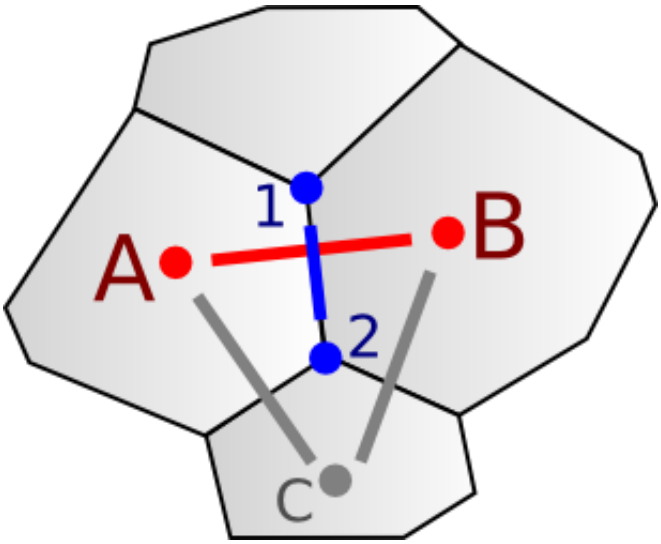
These biomes look good in the map generation demo, but each game will have its own needs. Realm of the Mad God for example ignores these biomes and uses its own (based on elevation and moisture).

Noisy Edges

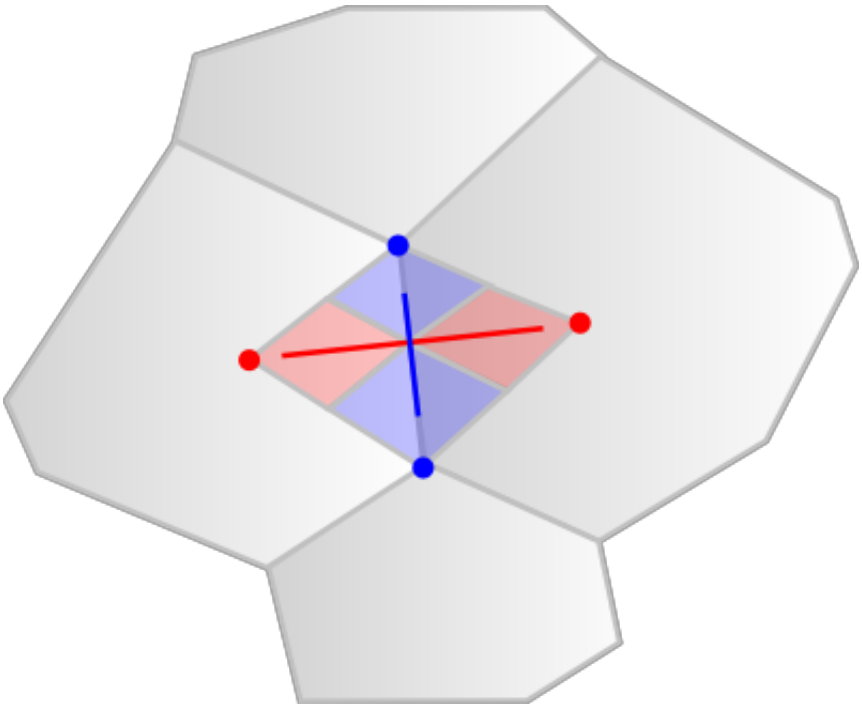
For some games, the polygonal maps are sufficient. However, in other games I want to hide the polygon structure. The main way I do that is to replace the polygon borders with a noisy line. Why

would I want a polygon structure if I'm just 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:



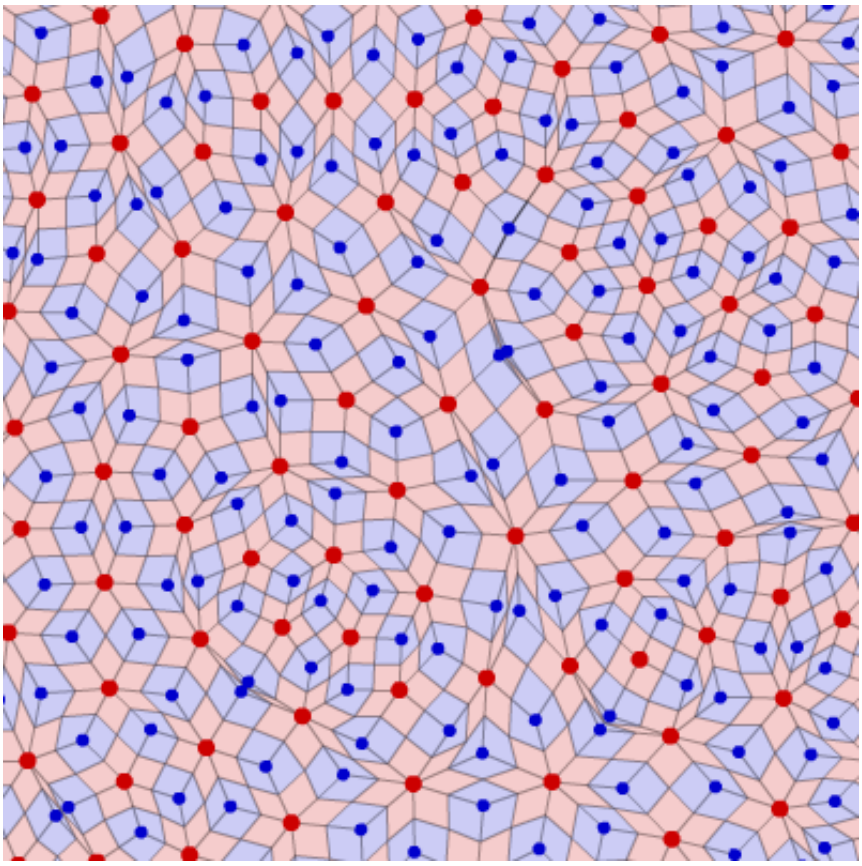
I wanted to make both types of line noisy without making them cross lines from other polygons. I also wanted to make them as noisy as feasible. I realized that points `A`, `1`, `B`, and `2` form a quadrilateral, and I could constrain the wanderings of the line segment to that quadrilateral:



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

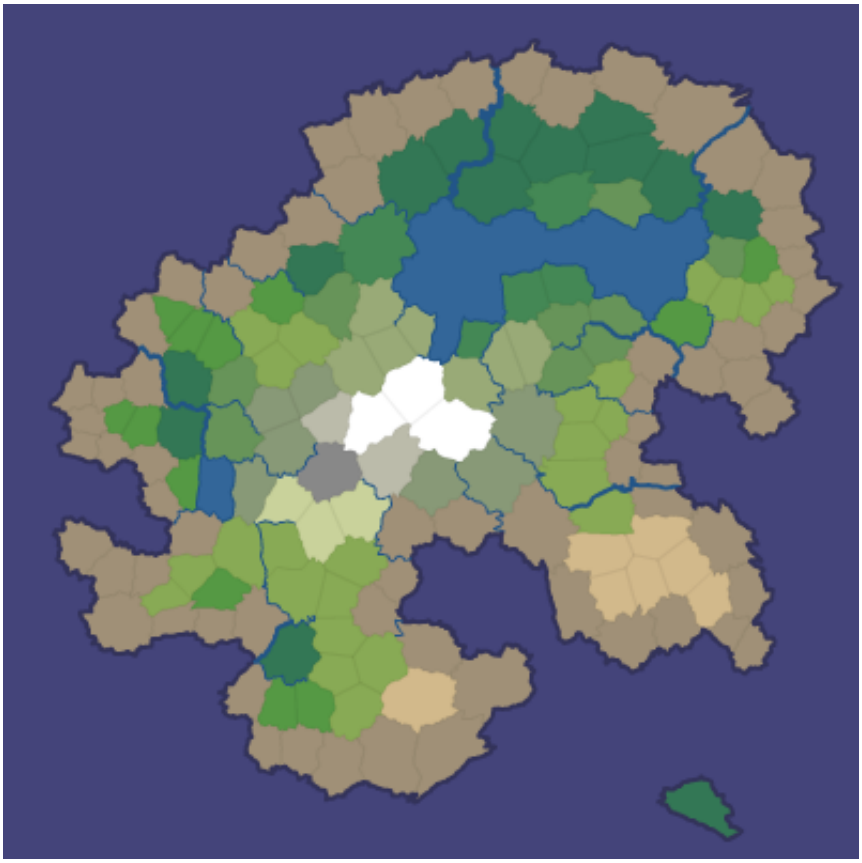
that the quadrilateral may not be convex; to divide it properly, I divide it at the midpoint of the Voronoi edge instead of at the intersection of the Voronoi and Delaunay edges.

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:



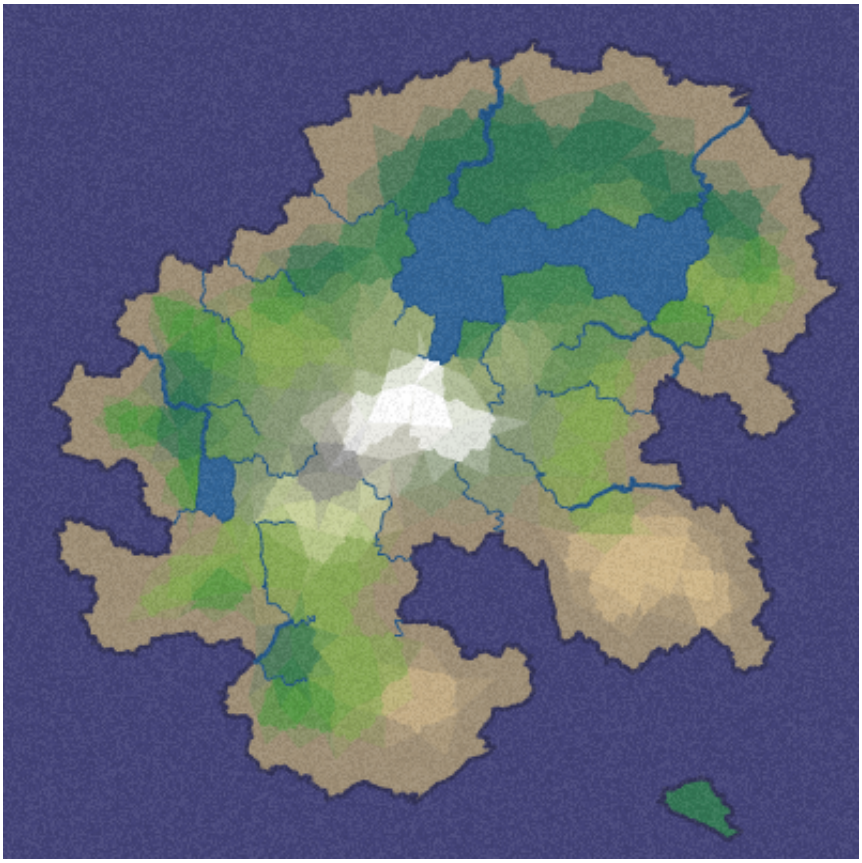
There are three places to tune the noisiness:

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.

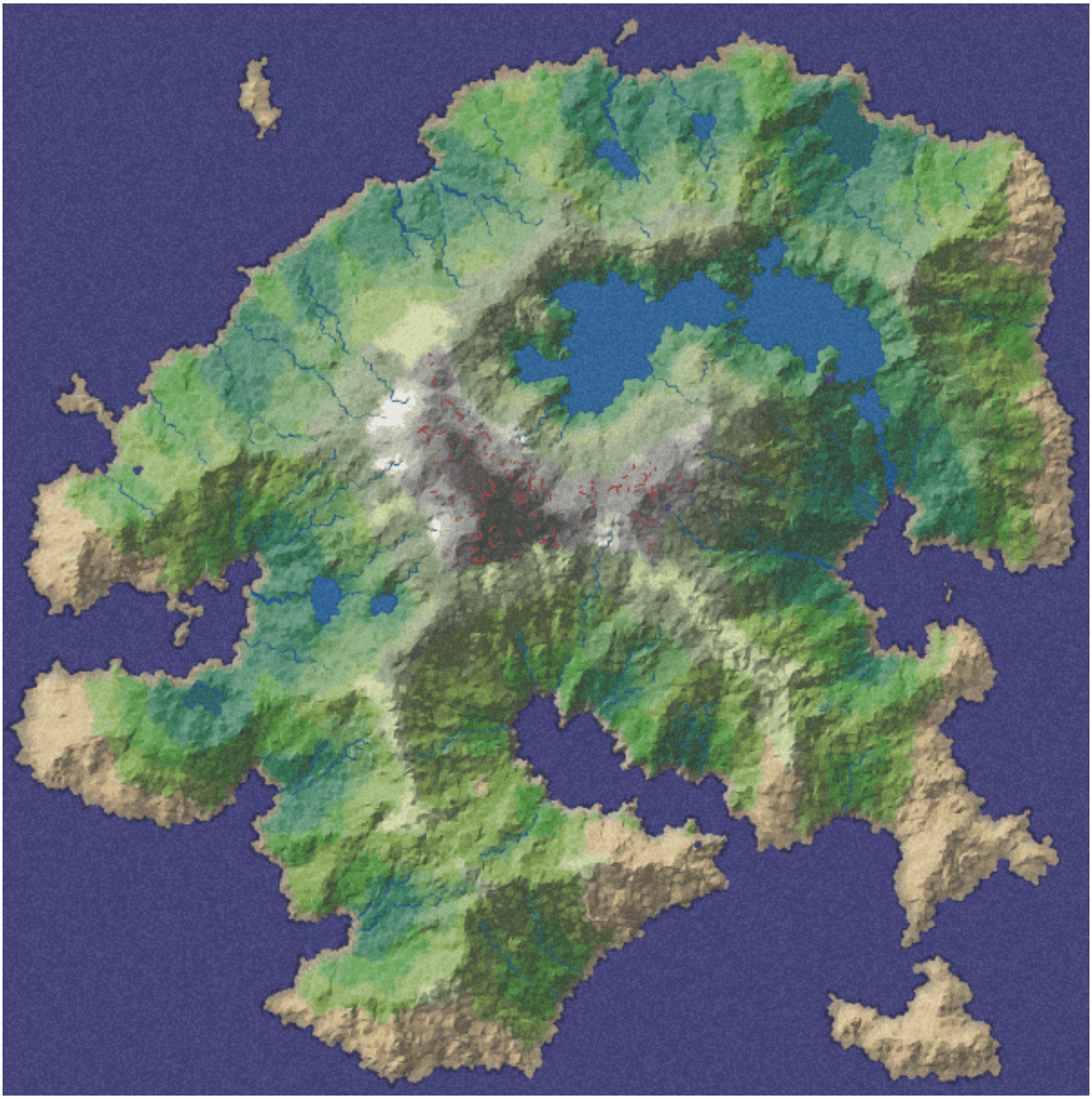
Noisy edges turn out to have a large impact on the map appearance, especially for rivers and coastlines.

More noise

I'm generally a fan of noise in game art, 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 just 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:

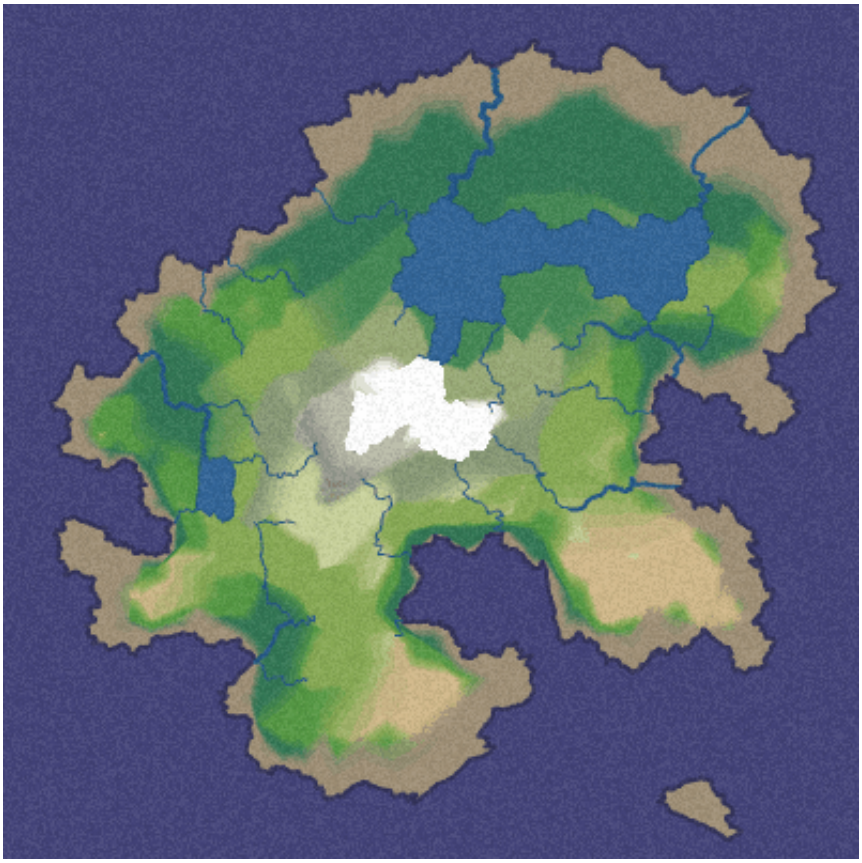


Here's a rendering with 16,000 polygons, noisy edges, a noise texture overlay, and simple lighting:



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:



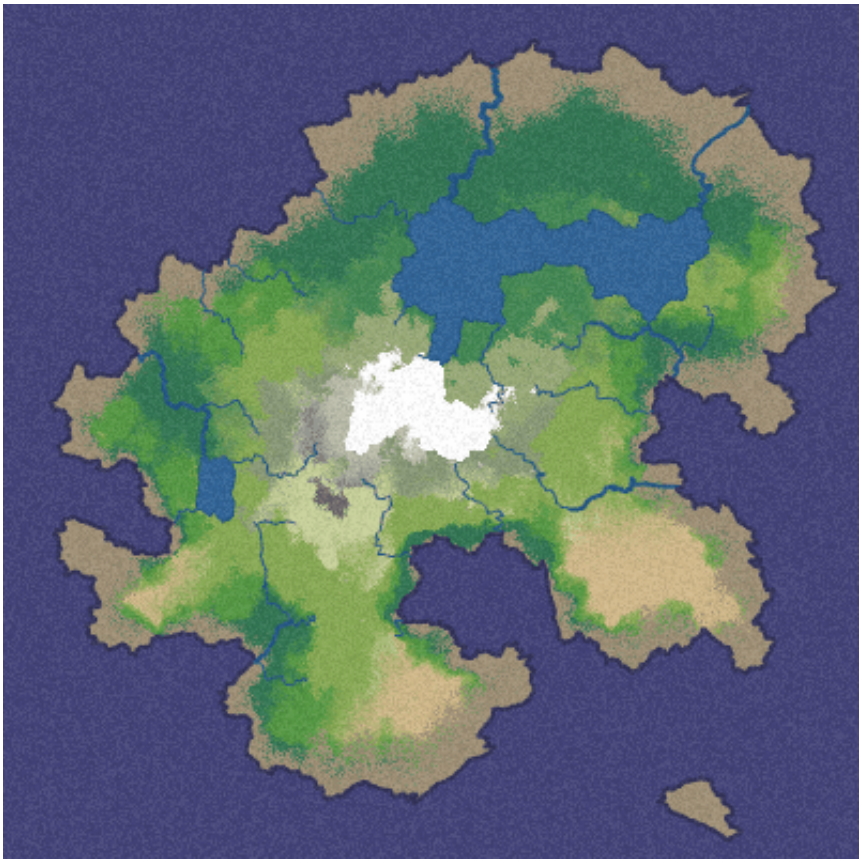
If the game doesn't need an entire polygon to be the same biome, this approach can be useful for making more interesting boundaries.

Distorted biome transitions

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

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

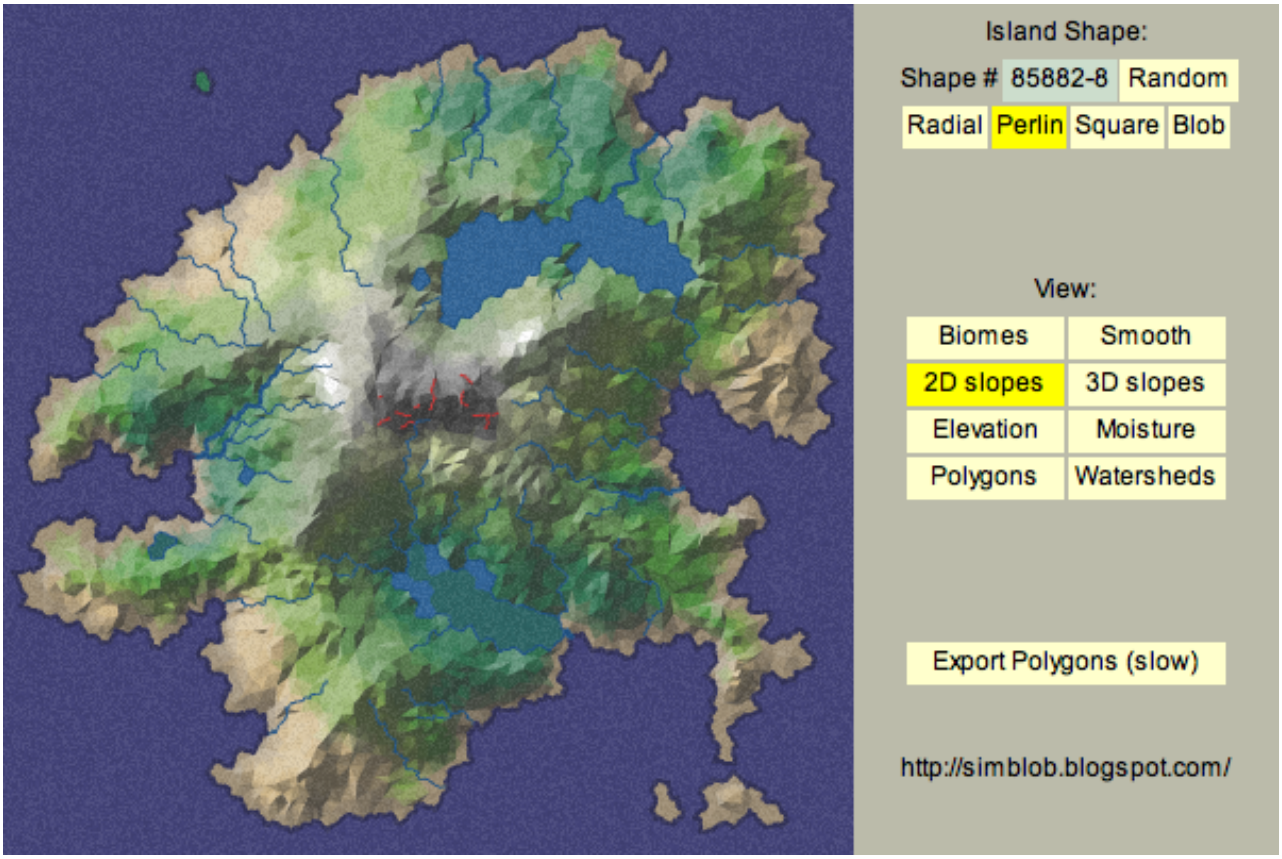
Here's an example of what this can do:



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.

Demo

I wrote a Flash demo to explore the generated maps:



The simplest way to explore the maps is to click Random and the various View options.

Try the demo!

In a shape number like `85882-8`, 85882 chooses the overall island shape and 8 is the random number seed for the details (random points, noisy edges, rivers). You can type in a shape number and press Return to generate that map. The demo also shows one of several road algorithms, and an unfinished watershed detection algorithm which might be useful for assigning territorial control.

Source

I've placed the Actionscript source under the MIT license; it's available on github. 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, and the code can go much higher, although I've not tried above 16,000. 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!

Jeff Terrace has contributed patches and also wrote a utility to convert the XML into COLLADA. Alex Schröder has been working on a Perl version that generates SVG output. Christopher Garrett has written a Voronoi/Delaunay port for iOS. Christophe Guebert has written a Java+Processing version. Baran Kahyaoglu has written a C#/ .NET version (source). Chris Herborth has ported Baran's code to Unity. At some point I'd like to port the code to Haxe, so that I can generate Actionscript, Javascript, C++, and Java versions.

The map generator wasn't designed for use in campaign maps but Welsh Piper (encounter tables, Minocra) and Kingdoms in Trevail are using the map generator to create campaign maps.

Appendix: More map features

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.

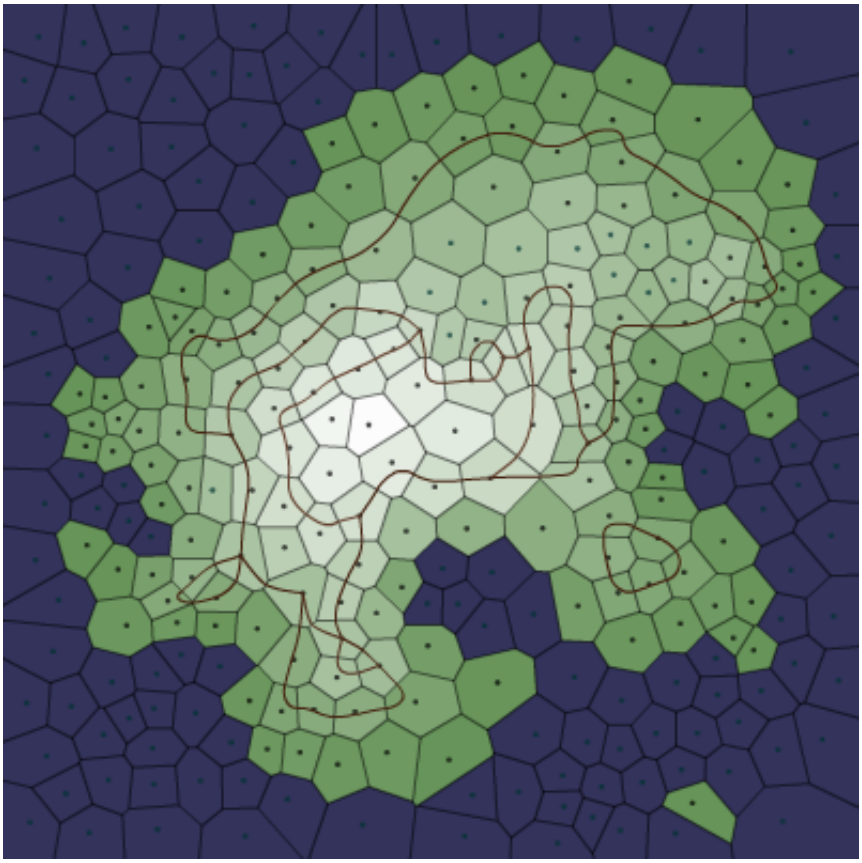
Where core map code can reference `edge.river` as a core field, the module can't do that. Instead, the module references its local variable `road[edge.index]`. This works for polygon centers and corners as well. It keeps the core clean.

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

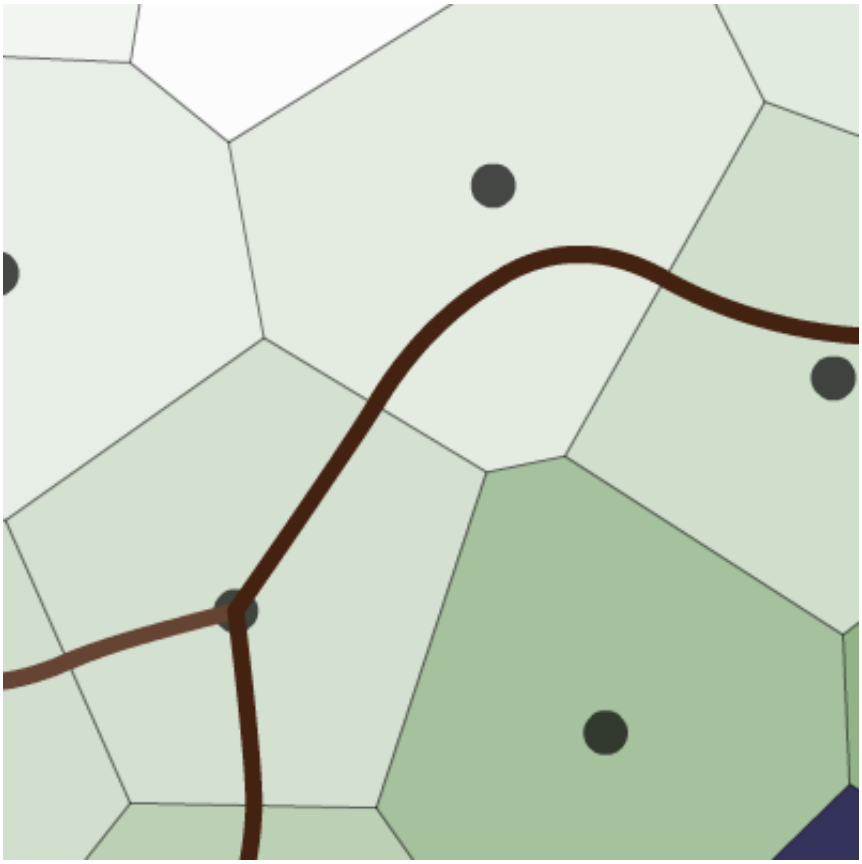
Roads

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:



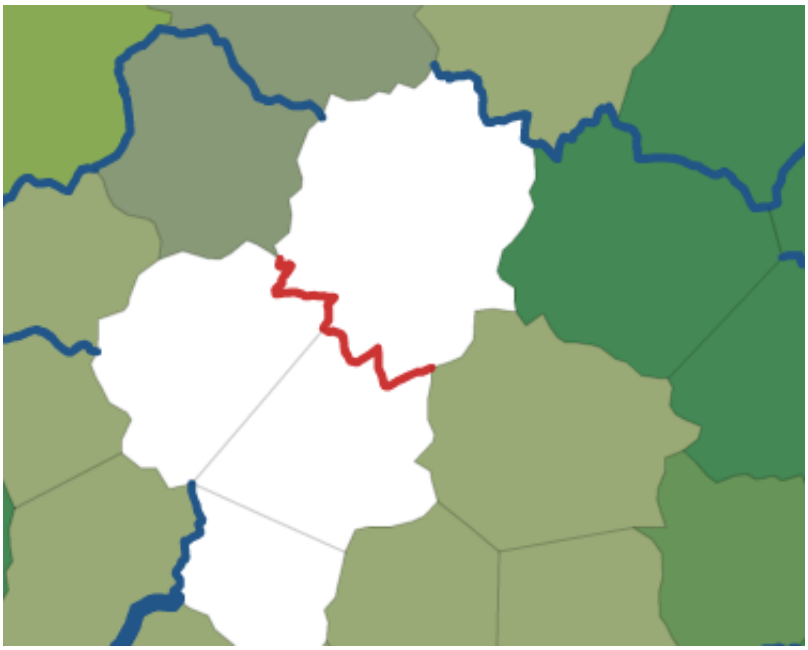
Whereas rivers meander along Voronoi edges (blue lines in the diagram above), roads go on Delaunay edges (red lines). Roads don't get the noisy edge treatment. Instead, they are drawn with splines from edge midpoint to midpoint:



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.

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:



Appendix: Future possibilities

Abstract Rendering

A *map* should show the relevant portions of the world, not all the details. With this projects 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).

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. 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.

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 for ideas on how to make the Voronoi regions interesting for gameplay.

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.

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.

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 instead of Lloyd's Algorithm.

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.

Appendix: Process improvements

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 noise for maps, so I tried it, with good results. 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. It was great! Most of the core system was in place in just 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.

Appendix: References

Thanks to the Dungeon League blog for a great series on procedural map generation, the Procedural Content Generation wiki for ideas for map generation, the incomplete Voronoi wiki for some useful resources about Voronoi diagrams.

My thanks to `nodename` for as3delaunay. It's an Actionscript 3 library for generating Voronoi and Delaunay graphs. Also my thanks to `polygonal` for the PM_PRNG 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 to optimize the PNG images on this page.

Delaunay triangulation and Voronoi polygons are taught in many graphics classes. For example, see Stanford CS 368(PDF). I also looked at relative neighborhood graphs and Gabriel graphs but didn't use either.

Delaunay triangulation can be used for maps, in the form of triangular irregular networks. Voronoi diagrams are also used for maps.

Fortune's Algorithm is one of several algorithms that can turn a set of points into Voronoi polygons. It's implemented in `as3delaunay`. Lloyd Relaxation 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 and edge representation, as well two pages that helped me with river generation: rivers and watersheds and crust and skeleton.

The Relief Shading Website has some images of shaded relief maps, 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 and my blog post and this article.

The Whittaker diagram is one way to predict common biomes given climate. Wikipedia has a page listing biome classification schemes and various biomes.

Wikipedia also has a nice list of landforms that one might want to generate in a game map generator. I did not explore these for this project.

Ken Perlin is the master of noise, with Perlin Noise. He's also made the much lesser known Simplex Noise(PDF). For this project I used Perlin Noise for the overall island shape. I was looking for blue noise, which led me to Dunbar and Humphreys's paper on noise generation and recursive Wang tiles before I found Lloyd's algorithm for better random point distribution. I also looked at Craig Reynold's textures but didn't have time to do anything with them.

Also interesting were these papers about generating worlds, Gozzy's wilderness map generator, donjon's world generator, a paper on procedural road generation, a paper on procedural city generation. Straight skeletons 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 is pretty neat.

Email me at amitp@cs.stanford.edu, or post a public comment: