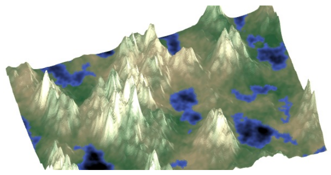
Making maps with noise functions

redblobgames.com/maps/terrain-from-noise

Table of Contents

One of the more popular pages on my site is about polygonal map generation. Making those maps was a lot of work. I didn't start there. I started with something *much* simpler, which I'll describe here. The simpler technique can make maps like this in under 50 lines of code:



I'm not going to explain how to *draw* these maps; that's going to depend on your language, graphics library, platform, etc. I'm only going to explain how to *fill an array* with height and biome map data.

Noise#

A common way to generate 2D maps is to use a bandwidth-limited noise function, such as Perlin or Simplex noise, as a building block. This is what the noise function looks like:

We assign each location on the map a number from 0.0 to 1.0. In this image, 0.0 is black and 1.0 is white. Here's how to set the color at each grid location in C-like syntax:

```
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    double nx = x/width - 0.5, ny = y/height - 0.5;
    value[y][x] = noise(nx, ny);
  }
}
```

The loop will work the same in Javascript, Python, Haxe, C++, C#, Java, and most other popular languages, so I'll show it in C-like syntax and you can convert it to the language you're using. In the rest of the tutorial, I'll show how the loop body (the `value[y][x]=...` line) changes as we add more features. At the end, I'll show a complete example.

Depending on the library you use, you may have to shift or multiply the values you get back to fit into the 0.0 to 1.0 range. Some libraries return 0.0 to 1.0; some return -1.0 to +1.0; some return other ranges like -0.7 to +0.7. Some don't say what they return, so you may have to look at the return values to figure out the range.

Elevation#

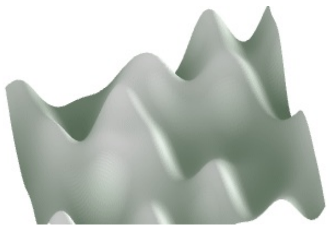
Noise by itself is just a bunch of numbers. We need to assign *meaning* to it. The first thing we might think of is to make the noise correspond to elevation (also called a "height map"). Let's take the noise from earlier and draw it as elevation:

The code is almost the same, except for what's inside the inner loop; it now looks like this:

```
elevation[y][x] = noise(nx, ny);
```

Yes, that's it. The map data is the same, but now I call it `elevation` instead of `value`.

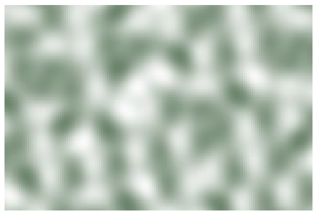
Lots of hills, but not much else. What's wrong?



Frequency

Noise can be generated at any *frequency*. I've only picked one frequency so far. Let's look at the effect of frequency. **Try moving the slider** to see what happens at different frequencies:

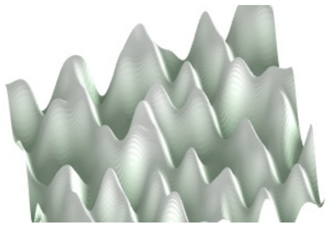
freq=



It's just zooming in and out. That doesn't seem very useful at first glance, but it is. I have another tutorial that explains the *theory*: things like frequency, amplitude, octaves, pink and blue noise, etc.

```
elevation[y][x] = noise(3.75 * nx, 3.75 * ny);
```

It's sometimes useful to think of *wavelength*, which is the inverse of frequency. Doubling the frequency makes everything half the size. Doubling the wavelength makes everything twice the size. The wavelength is a distance, measured in pixels or tiles or meters or whatever you use for your maps. It's related to frequency: `wavelength = map_size / frequency`.



Octaves

To make the height map more interesting we're going *add noise at different frequencies*:

+ + =

```
elevation[y][x] =    1 * noise(1 * nx, 1 * ny);
               +  0.5 * noise(2 * nx, 2 * ny);
               + 0.25 * noise(4 * nx, 2 * ny);
```

Let's mix big low frequency hills and small high frequency hills into the same map. **Move the slider** to add smaller hills to the mix:

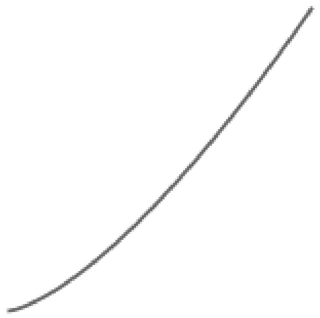
Now that looks a lot more like the fractal terrain we want! We can now get hills and rugged mountains, but we still don't get flat valleys. We need something else for that.

Redistribution

The noise function gives us values between 0 and 1 (or -1 and +1 depending on which library you're using). To make flat valleys, we can *raise the elevation to a power* . **Move the slider** to try different exponents.

exp=

```
e =    1 * noise(1 * nx, 1 * ny);
  +  0.5 * noise(2 * nx, 2 * ny);
  + 0.25 * noise(4 * nx, 4 * ny);
elevation[y][x] = Math.pow(e, 1.45);
```



Higher values *push middle elevations down into valleys* and lower values pull middle elevations up towards mountain peaks. We want to push them down. I use power functions here because they're simple, but you can use any curve you want; I have a fancier demo here.

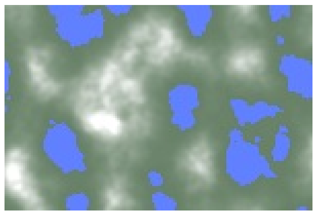
Now that we have a reasonable elevation map, let's add some biomes!

Biomes#

Noise gives us numbers but we want a map with forests, deserts, and oceans. The first thing to do is to make low elevations into water:

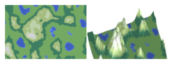
water=

```
function biome(e) {
    if (e < 0.39) return WATER;
    else return LAND;
}
```



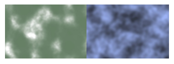
Hey, that's starting to look like a procedurally generated world! We have water, grass, and snow. What if we want more things? Let's make the sequence water, beach, grassland, forest, savannah, desert, snow:

```
function biome(e) {
  if (e < 0.1) return WATER;
  else if (e < 0.2) return BEACH;
  else if (e < 0.3) return FOREST;
  else if (e < 0.5) return JUNGLE;
  else if (e < 0.7) return SAVANNAH;
  else if (e < 0.9) return DESERT;
  else return SNOW;
}
```



Terrain based on elevation only

Hey, looks cool! You'll want to change the numbers and biomes for your game. Crysis will have a lot more jungles; Skyrim will have a lot more ice and snow. But no matter what you change the numbers to, this approach is a bit limited. The terrain types line up with the elevations, so they form bands. To make it more interesting, we need to choose biomes with something other than elevation. Let's create a *second* noise map for "moisture":



Elevation noise on left; moisture noise on right

Now let's use *both* elevation and moisture. In the diagram on the left below, the y-axis is the elevation (first diagram above) and the x-axis is the moisture (second diagram above). It produces a reasonable looking map:

Terrain based on two noise values

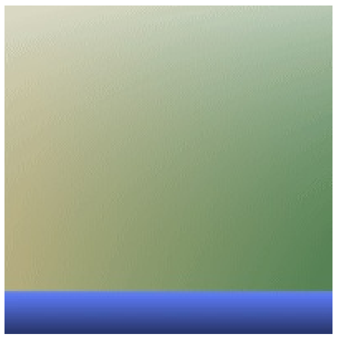
Low elevations are oceans and beaches. High elevations are rocky or snowy. In between we get a wide range of biomes. The code looks like this:

```
function biome(e, m) {
  if (e < 0.1) return OCEAN;
  if (e < 0.12) return BEACH;

  if (e > 0.8) {
    if (m < 0.1) return SCORCHED;
    if (m < 0.2) return BARE;
    if (m < 0.5) return TUNDRA;
    return SNOW;
  }

  if (e > 0.6) {
    if (m < 0.33) return TEMPERATE_DESERT;
    if (m < 0.66) return SHRUBLAND;
    return TAIGA;
  }

  if (e > 0.3) {
    if (m < 0.16) return TEMPERATE_DESERT;
    if (m < 0.50) return GRASSLAND;
    if (m < 0.83) return TEMPERATE_DECIDUOUS_FOREST;
    return TEMPERATE_RAIN_FOREST;
  }

  if (m < 0.16) return SUBTROPICAL_DESERT;
  if (m < 0.33) return GRASSLAND;
  if (m < 0.66) return TROPICAL_SEASONAL_FOREST;
  return TROPICAL_RAIN_FOREST;
}
```

You'll want to change all of those numbers to match the needs of your own game.

Alternatively if you don't need biomes, smooth gradients (see this article) can produce colors:

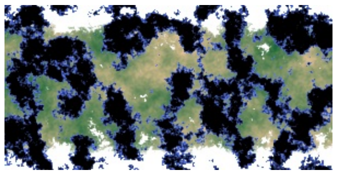
With either biomes or gradients, one noise value doesn't produce enough diversity, but two is pretty good.



Climate#

In the previous section I used *elevation* as a proxy for *temperature*. Higher elevations have lower temperatures. However, latitude also affects temperatures. Let's use both elevation and latitude to control temperature:

equator: hot cold
poles: hot cold



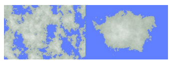
Near the poles (high latitudes) the climate is colder, and on the tops of mountains (high elevations) the climate is also colder. I haven't done much with this yet. For this demo I used `e = 10*e*e + poles + (equator-poles) * sin(PI * (y / height))` but *I don't think this is the best approach* and I think you'll need to experiment with formulas and tweak the parameters to make this work the way you want.

There's also a *seasonal* variation of climate. In summer and winter the northern and southern hemispheres get warmer and colder, but the equator doesn't change as much. There's lots more that can be done here, such as modeling prevailing wind and ocean currents and the biome's effect on climate and the ocean's moderating effect on temperatures.

Islands#

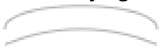
For some projects I want the boundaries of the map to be water. One way to do this is to generate a map as above and then *reshape* it.

How does this work? In a side view, the original noise terrain fits into a containing rectangle . To make islands, we *reshape* the container into something like . This pushes the middle up onto land and the edges down into water.


Before and after reshaping

The simplest shape is , formed using `e = (1 + e - d) / 2` where 0 ≤ `d` ≤ 1 is the distance from the center (either Manhattan or Euclidean or a mix). This shape guarantees the middle is on land and the edges are in water. It's overly aggressive but it's a simple place to start.

If you want to design your own shape that matches what you want from islands, you can define two shape functions. Choose a *lower* shape to push the map *up* and an *upper* shape to push the map *down*. These shapes are functions from distance `d` to elevation 0-1. Once you have designed these shapes, reshape the noise into an island using `e = lower(d) + e * (upper(d) - lower(d))`. You'll have to experiment to find functions you like.

In the lower shape doesn't push at all, so the middle of the map is not guaranteed to be on land. This allows more of the underlying Perlin/Simplex noise to show through. The upper shape pushes down anything away from the middle of the map. In  the upper shape is less aggressive, pushing down only near the edges, and the lower shape mildly pushes upwards near the middle. There are many other shapes to try!

Why stick to standard mathematical functions at all? As I explored in my article on RPG damage, everyone (including me) uses mathematical functions like polynomials, exponentials, etc., but on a computer we're not limited to those. We can draw *any* shape and use it here. Put the lower and upper shape in lookup tables and use them in your `lower(d)`, `upper(d)` functions.

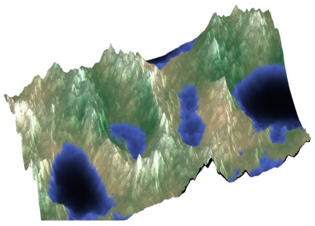
Ridged noise#

Instead of raising the elevation to a power, we can use absolute value to create sharp ridges:

```
function ridgenoise(nx, ny) {
  return 2 * (0.5 - abs(0.5 - noise(nx, ny)));
}
```

To add octaves, we can vary the amplitudes of the higher frequencies so that only the mountains get the added noise:

```
e0 =    1 * ridgenoise(1 * nx, 1 * ny);
e1 =  0.5 * ridgenoise(2 * nx, 2 * ny) * e0;
e2 = 0.25 * ridgenoise(4 * nx, 4 * ny) * (e0+e1);
e = e0 + e1 + e2;
elevation[y][x] = Math.pow(e, exponent);
```

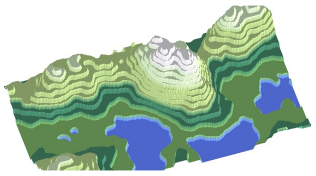


n=8

I don't have much experience with this technique and will have to play with it more to learn how to use it well. It might also be interesting to mix ridged low frequency noise with non-ridged high frequency noise.

Terraces#

If we round the elevation to the nearest of 12 levels we get terraces:



n=12

This is an application of elevation redistribution functions of the form `e = f(e)`. Earlier we set `e = Math.pow(e, exponent)` to make mountain peaks steeper; here we use `e = Math.round(e * 12) / 12` to make terraces. By using a function other than a step function the terraces can be rounder or only occur at some elevations.

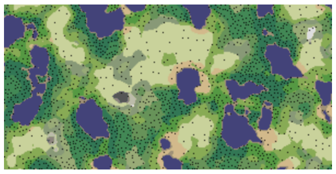
Tree placement#

We usually use fractal noise for elevation and moisture, but it can also be used for placing irregularly spaced objects like trees and rocks. For elevation we have higher amplitudes with lower frequencies ("red noise"). For object placement we want to use higher amplitudes with higher frequencies ("blue noise"). On the left is a blue noise pattern; on the right are the locations where the noise is greater than the nearby values:

R=3

```
for (int yc = 0; yc < height; yc++) {
  for (int xc = 0; xc < width; xc++) {
    double max = 0;
    // there are more efficient algorithms than this
    for (int yn = yc - R; yn <= yc + R; yn++) {
      for (int xn = xc - R; xn <= xc + R; xn++) {
        double e = value[yn][xn];
        if (e > max) { max = e; }
      }
    }
    if (value[yc][xc] == max) {
      // place tree at xc,yc
    }
  }
}
```

By choosing a different R for each biome we can get a variable density of trees:



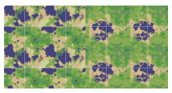
It's cool that Perlin/Simplex noise can be used to place trees, but other algorithms, such as jittered grids, Poisson discs, Wang tiles, or graphics dithering, are often more efficient and also produce a more even blue noise distribution.

Wraparound maps#

Sometimes we want the east edge of the map to match up with the west edge. This corresponds to a *cylinder* in 3d space. We can implement this with a minor change. We'll interpret the x value on the flat map as an *angle* in the cylinder world. Then we convert the angle to cartesian coordinates. To also make the north and south edges match up, we can apply the same pattern again to turn the y value into an angle, and look things up in a 4d noise space. Let's see how the maps look adjacent to copies of themselves:

The first one wraps east-west but not north-south. The second one wraps in all four directions. Here's some code:

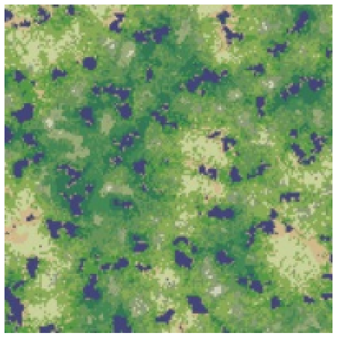
```
const TAU = 2 * M_PI;

function cylindernoise(double nx, double ny) {
    double angle_x = TAU * nx;
    /* In "noise parameter space", we need nx and ny to travel the
       same distance. The circle created from nx needs to have
       circumference=1 to match the length=1 line created from ny,
       which means the circle's radius is 1/2π, or 1/tau */
    return noise3D(cos(angle_x) / TAU, sin(angle_x) / TAU, ny);
}

function torusnoise(double nx, double ny) {
    double angle_x = TAU * nx,
           angle_y = TAU * ny;
    return noise4D(cos(angle_x) / TAU, sin(angle_x) / TAU,
                   cos(angle_y) / TAU, sin(angle_y) / TAU);
}
```


East+West only and
North+South+East+West wraparound

I've not experimented with this much yet. Take a look at Ron Valstar's guide for more explanation.

To infinity and beyond#

The calculation of the biome at position (x,y) is independent of calculations at any other position. This **local calculation** results in two nice properties: it can be calculated in parallel, and it can be used for infinite terrain. **Put the mouse over the minimap** on the left to generate a map on the right. We can generate any part of the map without generating (or having to store) the whole thing.



Implementation#

Using noise for generating terrain is a popular technique, and you can find tutorials for many different languages and platforms. The map generation code is pretty similar across languages. Here's the simplest loop, in three different languages:

- **Javascript:**

```
let gen = new SimplexNoise();
function noise(nx, ny) {
  // Rescale from -1.0:+1.0 to 0.0:1.0
  return gen.noise2D(nx, ny) / 2 + 0.5;
}

let value = [];
for (let y = 0; y < height; y++) {
  value[y] = [];
  for (let x = 0; x < width; x++) {
    let nx = x/width - 0.5, ny = y/height - 0.5;
    value[y][x] = noise(nx, ny);
  }
}
```

- **C++:**

```
module::Perlin gen;
double noise(double nx, double ny) {
  // Rescale from -1.0:+1.0 to 0.0:1.0
  return gen.GetValue(nx, ny, 0) / 2.0 + 0.5;
}

double value[height][width];
for (int y = 0; y < height; y++) {
  for (int x = 0; x < width; x++) {
    double nx = x/width - 0.5,
           ny = y/height - 0.5;
    value[y][x] = noise(nx, ny);
  }
}
```

- **Python:**

```
from opensimplex import OpenSimplex
gen = OpenSimplex()
def noise(nx, ny):
    # Rescale from -1.0:+1.0 to 0.0:1.0
    return gen.noise2d(nx, ny) / 2.0 + 0.5

value = []
for y in range(height):
    value.append([0] * width)
    for x in range(width):
        nx = x/width - 0.5
        ny = y/height - 0.5
        value[y][x] = noise(nx, ny)
```

Once you have a noise library, these are all pretty similar. Try opensimplex for Python or libnoise for C++ or simplex-noise for Javascript. There are lots of noise libraries for most popular languages. Alternatively, you may want to spend time studying how Perlin noise works, or implementing it yourself. *I didn't.*

Once you have found a noise library for your favorite language, the details will vary (some will return numbers from 0.0 to 1.0 and others from -1.0 to +1.0) but the basic idea is the same. For a real project you may want to wrap the `noise` function and the `gen` object into a class but those details aren't relevant here so I made them global.

For this simple project it doesn't matter that much whether you use Perlin noise, Simplex noise, OpenSimplex noise, value noise, midpoint displacement, diamond displacement, or an inverse Fourier transform. There are pros and cons of each of these but they all produce similar enough output for this type of map generator.

The *drawing* of the map is going to be platform-specific and game-specific so I'm not providing that; this code is just to generate the elevations and biomes, which you'll want to draw yourself in whatever style your game uses. Feel free to copy, port, and use it for your own projects.

Playground#

I've covered mixing octaves, raising the elevation to a power, and combining elevation and moisture to pick a biome. Here's an interactive diagram that lets you play with all of these parameters, and then shows how the code is put together:

Here's the code:

```
var rng1 = PM_PRNG.create(seed1);
var rng2 = PM_PRNG.create(seed2);
var gen1 = new SimplexNoise(rng1.nextDouble.bind(rng1));
var gen2 = new SimplexNoise(rng2.nextDouble.bind(rng2));
function noise1(nx, ny) { return gen1.noise2D(nx, ny)/2 + 0.5; }
function noise2(nx, ny) { return gen2.noise2D(nx, ny)/2 + 0.5; }

for (var y = 0; y < height; y++) {
  for (var x = 0; x < width; x++) {
    var nx = x/width - 0.5, ny = y/height - 0.5;
    var e = (0.68 * noise1( 1 * nx,  1 * ny)
           + 0.59 * noise1( 2 * nx,  2 * ny)
           + 0.23 * noise1( 4 * nx,  4 * ny)
           + 0.06 * noise1( 8 * nx,  8 * ny)
           + 0.07 * noise1(16 * nx, 16 * ny)
           + 0.09 * noise1(32 * nx, 32 * ny));
    e /= (0.68+0.59+0.23+0.06+0.07+0.09);
    e = Math.pow(e, 5.00);
    var m = (1.00 * noise2( 1 * nx,  1 * ny)
           + 0.71 * noise2( 2 * nx,  2 * ny)
           + 0.91 * noise2( 4 * nx,  4 * ny)
           + 0.08 * noise2( 8 * nx,  8 * ny)
           + 0.06 * noise2(16 * nx, 16 * ny)
           + 0.00 * noise2(32 * nx, 32 * ny));
    m /= (1.00+0.71+0.91+0.08+0.06+0.00);
    /* draw biome(e, m) at x,y */
  }
}
```

A tricky bit: you need to use different seeds for the elevation and moisture noise. Otherwise they'll end up being the same, and your maps won't look nearly as interesting. In Javascript I use the prng-parkmiller library; in C++ you can use two separate linear_congruential_engine objects; in Python you can instantiate two separate instances of the random.Random class.

Another tricky bit: once you combine multiple octaves of noise, the range of output may not be what you expected, and you may have to add/multiply the output value into the desired range (such as 0.0 to 1.0). Scott Turner has written a bit more about common issues with noise.

Thoughts#

What I like about this approach to map generation is that **it's simple**. It's fast. It's very little code to produce decent results.

What I don't like about this approach is that it's limited. Local calculation means every location is independent of every other location. Different areas of the map *don't relate to each other*. Every place on the map "feels" the same. There are no global constraints like "there should be between 3 and 5 lakes" or global features like a river flowing from the top of the highest peak down to the ocean. Another thing I don't like is that it takes a lot of tweaking of parameters to get something you like.

Why do I recommend it then? I think it's a good starting point, especially for indie games or game jams. Two of my friends wrote the initial version of Realm of the Mad God in 30 days, for a game competition. They asked me to help them make the maps. I used this technique (plus some extra features that turned out not to be that useful) to make a map for them. Months later, after getting feedback from players and looking at the game design a lot more, we designed the more advanced map generator using Voronoi polygons, described here. That map generator doesn't use the techniques from this page, but uses noise very differently to produce maps.

Noise-based elevation is fun and easy to get started with but you can quickly run into limits. Scott Turner has an insightful essay with reasons to use alternatives.

More#

There are *lots* of cool things you can do with noise functions. If you search the web you'll see variants such as turbulence, billow, ridged multifractal, amplitude damping, terraced, voronoi noise, analytical derivatives, domain warping, and others. Take a look at this page for inspiration. I'm not covering them here; instead I'm focused on simplicity for this article.

My previous map generation projects that influenced this one:

- I used generic Perlin noise for my first Realm of the Mad God map generator. We used that for the first six months of alpha testing, and then replaced it with a custom-designed Voronoi polygon map generator based on the gameplay needs we identified during the alpha testing. The biomes in this article and their colors come from those projects.
- While studying signal processing for audio, I wrote a tutorial on noise that covers concepts like frequency, amplitude, octaves, and "colors" of noise. The same concepts that work for audio also apply to noise-based map generation. I made some unpolished terrain generation demos at the time but never finished them.
- Sometimes I experiment to find limits. I wanted to see how little code I could get away with while still producing reasonable maps. In this mini-project I got down to **zero** lines of code— it's all image filters (turbulence, threshold, color gradients). I was both pleased and disturbed by this. How much of map generation can be done with image filters? Quite a lot. Everything in the "smooth gradient color scheme" I described earlier came from this experiment. The noise layer is a turbulence image filter; the octaves are images layered on top of each other; the exponent is called the "curves adjustment" tool in Photoshop.

It bothers me somewhat that most of the code we game developers write for noise-based terrain generation (including midpoint displacement) turns out to be the same as audio and image filters. On the other hand, it produces decent results with very little code, so that's why I wrote the article you're reading. It's a *quick & easy starting point*. I usually don't use these types of maps for long; I'll replace them with a custom map generator once more of the game is built and I have a better sense of what types of maps best match that game's design. That's a common pattern for me: start with something extremely simple, then replace it only after I better understand the system I'm working on.

There are a *lot* of cool things you can do with noise, and I've barely explored them here. Try out the Noise Studio to interactively explore lots of possibilities.