

Chapter 3

An Oscar-Winning Algorithm

Fractal Noise

Ken Perlin - 1983

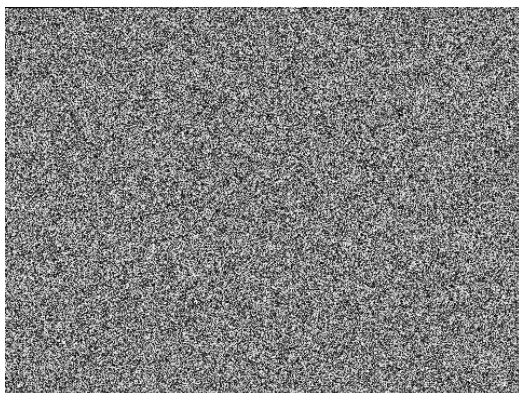
Written by Bernat Romagosa

3.1 Natural Noise for an Artificial World

In the mid eighties, Ken Perlin was working as system architect at a computer company that provided CGI for a movie called *Tron*. In the film, a bunch of hackers would jump into a virtual world and famously race each other on *light cycles*, futuristic-looking motorbikes that left a neon light trail as they blazed through the digital grid.

Even though the computer-generated graphics in the movie were beyond anything that had been produced up to then, Perlin was slightly annoyed by what textures looked like in the virtual world. Sure, that was supposed to look like an artificial world, but the textures looked a bit too . . . artificial!

Ken Perlin realized that, in the real world, many things that look random actually follow some sort of hidden pattern. There is some randomness in what mountain ranges look like, sure. Archipelagos also feel kind of random, as do clouds and the surfaces of rocks and trees. They're not totally random, though. This is what a totally random image looks like:



In this image, each pixel takes a random grayscale color between 100% black and 100% white. In computer terms, this type of image is known as *random noise*.

Natural textures have a bit of this noisiness in it, but without looking totally random. Take, for instance, this detail of some tree bark:



The texture is random, but there clearly is a structure to this randomness. Patterns can be made out, and they seem to follow some kind of geometric logic.

This is pretty much what Ken Perlin also noticed. By mimicking the way nature develops random-looking patterns, he was able to produce textures that resembled those of naturally occurring materials. This technique helped advance CGI in such a way that, in 1997, the Oscar's Academy of Motion Picture Arts and Sciences decided to honor him with an Award for Technical Achievement.

Ever since then, *Perlin noise* -and a bunch of derivative algorithms for naturally-looking noise- has been used to produce all kinds of seemingly unrelated objects and textures, including things like grass leaves,

hair, cloth, fantasy maps, water or smoke.

The algorithm we're going to implement in this chapter is not the one that Perlin devised, but one that inherits from his original ideas. Our algorithm belongs to the family of *fractal noise* generators, and it can take different names and produce different sorts of textures depending on the kind of noise we start off from and the way we decide to merge the different levels of randomness together.

To illustrate the general process, let's begin by looking at an image of a rock at different levels of zoom:



Notice how zooming into the texture seems to reveal different layers of randomness, just as if natural processes worked at different micro and macroscopic levels at the same time, resulting in an organic, almost

fractal pattern. The scale-like structures that can be clearly seen in the last picture, look pretty much like random noise in the first one.

Our noise algorithm will be base on this idea of combined differently sized random patterns and, although not officially Perlin noise, we will be using it to produce results very much like those we could obtain with Ken Perlin's original algorithms.

3.1.1 Make Noise

To make *fractal noise* we'll first need to, well, make some noise. In particular, we'll need to be able to produce random noise like the one we saw in the first figure of the previous section.

To produce random noise we'll need to choose a random value for each pixel in a grid. We're going to make the image as big as the default Snap! stage, that is, 480×360 pixels. As you may already imagine, there is an iterative, a functional and a hyper- way of doing so.

The iterative way comes rather natural to those of us who've been unlucky enough to be trained as computer scientists. That is, those of us who've been made to think like a computer. The idea is to add a huge amount of random numbers between 0 and 255¹ to a list, and then treat that list as a grid of 480×360 pixels.



The code above begins by creating an empty list. Then, 480×360 times, it adds a random number between 0 and 255 to it. It then ends

¹Brightness values for the red, green and blue channels of a pixel range from 0 to 255. When we tell Snap! to treat a one-dimensional list as a costume (that is, a list of individual numbers, as opposed to a list of three-element lists) it will map each value equally to the red, green and blue channels of each pixel, resulting in a grayscale image.

by creating a new costume that's 480 pixels wide by 360 pixels high, and switching to it so we can see it worn by the current sprite on stage.

This does work, and it's not even terribly slow, but we can do better.

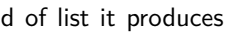


The functional way abstracts away the process of creating that list and repeatedly adding a random value to it.

Notice how we're using the  ² block to quickly obtain a list of our desired size. We're then repopulating it by mapping  over it.

This way also works and is faster than the iterative one. However, as usual, the hyper- way, albeit with some more mind-bendingness required, produces the result almost instantly and reads beautifully.



Let's take a moment to read this short snippet and fully understand why it works. Maybe try to run the  part first, see what kind of list it produces, then add the  part and see what it does. Maybe try it with differently-dimensioned lists. Try to play with different values and see how the result changes.

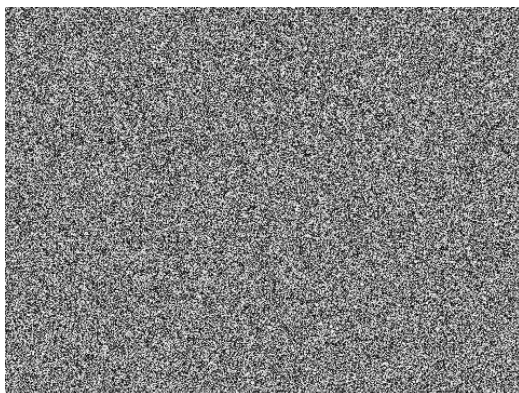
The reason this works is that the  block cre-

²The  block takes a list and adapts its dimensions to the specified sizes. In this case we're taking a single element, "0", and reshaping it into a one-dimensional list of 480×360 elements.

ates a two-dimensional list -or matrix- of 480×360 zeroes.³ Then,  populates that list with numbers that go from the number in the original list -that is, zero- to the value in the other input of the  block -that is, 255-.

It's been years since Snap! introduced support for hyper operations, but we keep marveling over their expressiveness and swiftness. It takes a bit to get used to this way of thinking, but it unfolds such expressive and computational power that we think every Snap! user should try to embrace the hyper- way.

In any case, all of the methods above should have produced a perfectly useful noise costume such as this one:



Now we've got some computer-generated (and computer-looking) noise, so we can begin the transformation process to turn this artificial

³You may have expected the order of these dimensions to go the other way around, as in 480 should go before 360. However, in the  block, dimensions are specified from the outside in. That is, the first number represents the outermost dimension, and the last number represents the innermost one. Since we need to obtain a matrix of 360 rows containing 480 elements each, the order needs to read 360, then 480.

grid of pixels into something that nature itself could have produced.

3.1.2 Layers of Noise

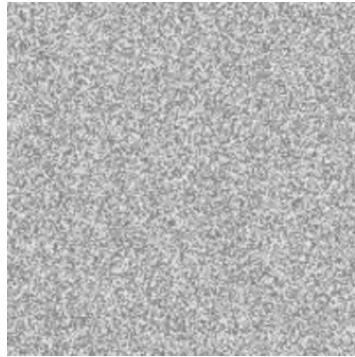
In the introduction to this chapter we hinted at the different macro- and microscopic levels at which nature seems to sprinkle its randomness. Since the pixel is the tiniest possible visible unit that we can generate on a computer screen, what we've created in the last section could be considered the equivalent of nature's microscopic level of noise.

How are we going to produce more macroscopic levels, though? A naive idea would be to just scale up these pixels by a factor of two, and then merge them together, but this feels like it shouldn't really work. After all, the resulting scaled up noise is going to look blocky, and maybe even blurry because of how computers apply zoom. This doesn't sound like it would produce anything like the result of a natural process.

But naive ideas are usually easy to try, so why not give it a shot?

First of all, we need to make sure that the current sprite is wearing a random noise costume like the ones we produced in the previous section, and that it is centered on the stage. Now, to merge two sizes of a costume together we could try to just stamp it on the stage, then increase its size by 100% and stamp it again, but if we try this the second stamping is just going to replace the first one. That's why we'll need to first adjust the sprite opacity, which we can do by using the  block. A value of 75% should make it transparent enough for several layers to merge together and still have an individually meaningful impact to the resulting image.

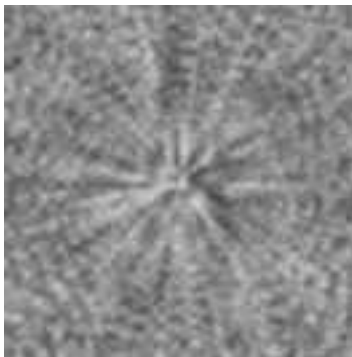
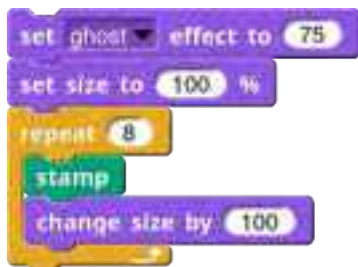
As a first quick test, let's try it with just two stampings.



This looks somewhat promising. Let's try to do the same thing a bunch more times and see what comes out of it. But first, remember to clear the stage and reset the sprite costume to be the random noise one:

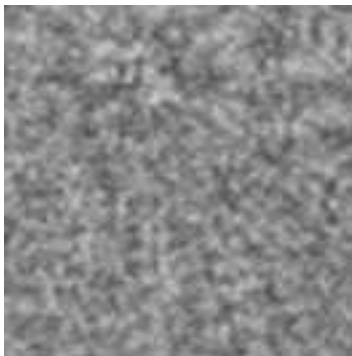


To repeat the process a bunch of times, we can just wrap the stamp and resize part in a loop. Let's try to do it 8 times, for instance.

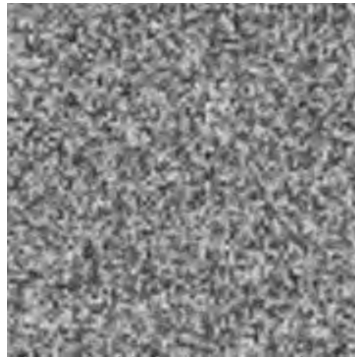


That's too sad. The naive method did not work. Of course, when zooming into the costume we ended up generating a tunnel effect. Probably because all the costume stampings share a common center. That's what must make the effect not so apparent in the corners, since they're the farthest from the center.

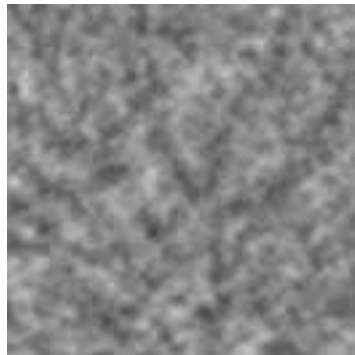
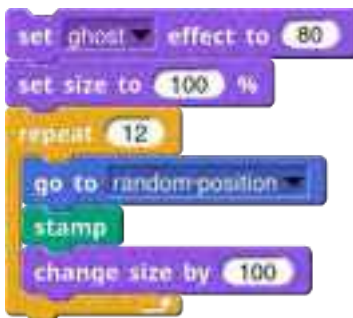
What if, naive as we still are, we just jumped to a different random point before stamping each layer? Again, remember to clear the stage and wear back the random noise costume first.



Lo and behold, that's a good enough example of *fractal noise*. Random looking, yet smooth and somewhat organic. Even though the two algorithms are quite different, the fractal nature of the process actually makes it look like *fractal Perlin noise*. Playing with different values for the ghost effect and the amount of repetitions -called *octaves* in Perlin slang- will produce different variations. Less octaves and less transparency will result in a cork-like texture.



While more octaves and more transparency will generate cloud-like images.



Sometimes life is fair to those of us who stay naive, but beware, naiveness only gets you so far. If you try to produce too many octaves you'll end up with a costume so big that your browser tab will run out of memory and Snap! may even crash.

3.2 A Noise Generator

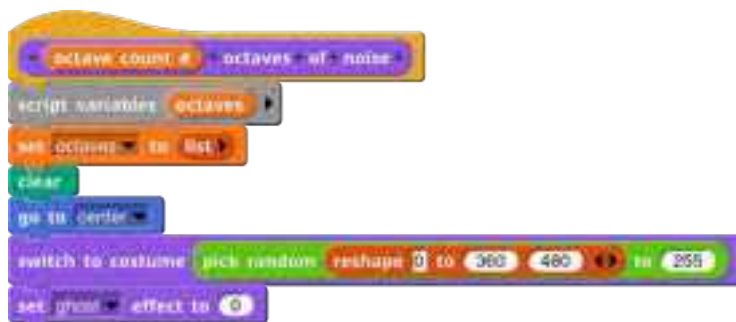
Now that we understand how to make organic-looking noise, it's a good idea to spend a few moments abstracting what we've learned so far into a block that takes an octave count input and produces a fractal noise costume.

While we're at it, we're going to use a neat trick that will solve the memory-eating problem and greatly increase the looks of our noise. The idea is very simple: we'll first generate all the noise layers and store them into a list. Only after having generated the list of different octaves we're going to merge them together. We'll see how and why this is going to solve the memory problem later.

But before we start, a word of caution for the functional programming purists out there: we're going to build a couple of reporter blocks that *mutate the state* of the sprite. Not only that, but they're also going to *mutate the state* of the stage. You don't need to worry, though. The burden for this sin lays on our shoulders, and we're going to make up for it further down the road.

So, with that out of the way, let's begin by creating a new reporter block in the *Looks* category that produces a list of random noise images at different octaves. We will start by having it declare a local variable and initializing it to an empty list. Then, we'll proceed to clear the stage, center the sprite on the stage, and wear a random noise costume.

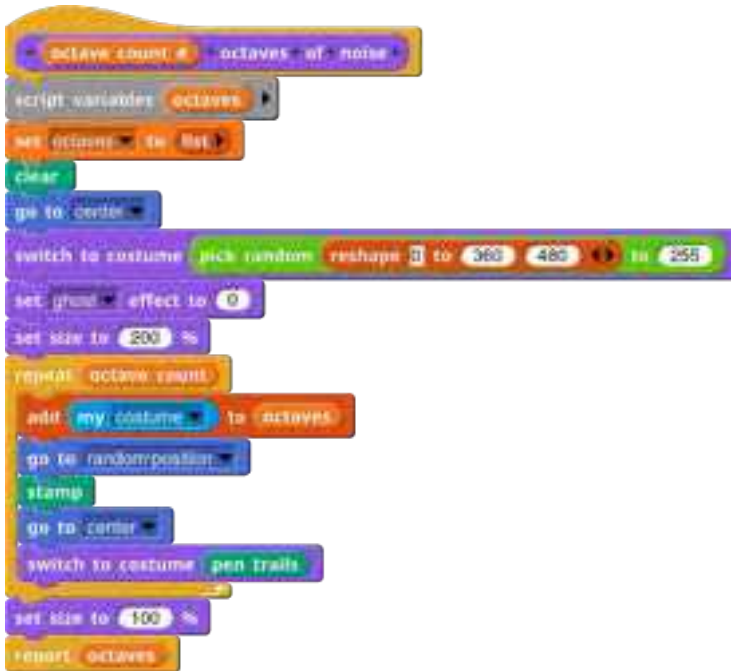
In preparation for merging, where we'll be messing with the ghost effect, we also want to make sure to set it to 0 before generating the noise octaves.



Now, for each octave, we'll want to store the current costume into the *octaves* list and generate the next octave.

And here's where the neat trick comes in: if we keep stamping the last layer onto the stage and wearing the just stamped pen trails as a new costume we'll only ever need to deal with a zoom level of 200%.

As an added bonus, this will also make sure that the resulting noise layers grow at a rate of $2\times$, which is what most of the literature for fractal noise algorithms advise. Before, we didn't dare scaling the costume exponentially, as we'd have eaten the memory in just a few iterations, but now every layer takes the exact same amount of memory to compute, so we can really experiment with the octave count without fear.



After stamping each octave we need to remember to go back to the center of the stage before wearing the pen trails, else they will keep being offset by the current sprite position and we'll most probably end up off-stage and wearing an empty costume.

If we run this block for, say, eight octaves, we'll get a list of differently zoomed layers of noise ready to be merged together.



Most noise algorithms require some sort of smoothing or blurring to be applied to the different layers, but we're getting this for free as a side effect of scaling the images in Snap!. The graphics card is doing that for us automatically. For other applications, this could be a problem, but in this occasion we surely welcome the GPU's eagerness to do more than we're asking for!

Merging these layers together is now rather trivial. We just iterate over each costume, stamp it and wear the resulting pen trails. Since we're only ever merging two costumes together, and wearing the result of stamping one on top of the other, the ghost effect can always stay at 50%.

The new fractal noise block looks like this:



And the stunningly beautiful noise it generates looks like this:



The noise we generated with the purely naive method was a pretty good example of fractal noise, but the one we've just made is a perfect textbook example.

3.2.1 A Functionally Acceptable Method

The previous method works and yields rather satisfying fractal noise, but it also breaks one of the sacred laws of functional programming:

*“Thy reporters shalt **not** mutate any state whatsoever.”*

It could seem picky, but this law isn’t sacred *just because*. Having reporters that mutate your state creates all sorts of problems where you end up having to write uglier code because you can’t place those reporters where they should be. You may be using the fractal noise reporter in a loop where you want a sprite to draw on stage according to the values of your noise, and when the reporter gets executed it ends up clearing the stage and switching your sprite’s costume.

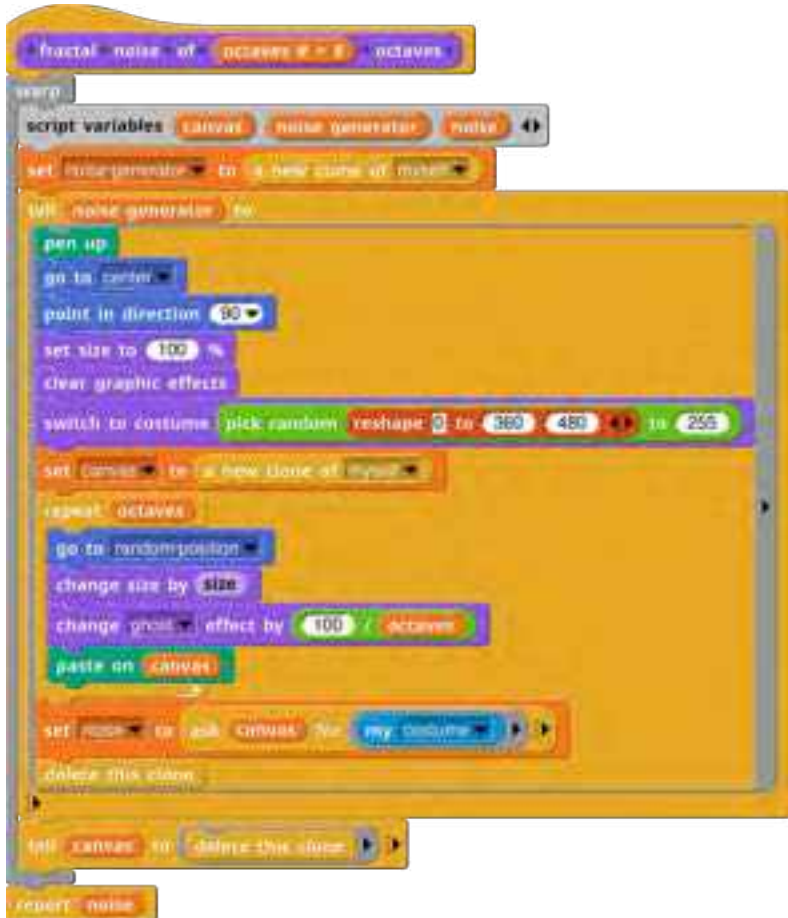
To solve this we’ll follow pretty much the exact same algorithm, but using two temporary clones to isolate the process. One of the clones is going to be generating the noise octaves, and the other one is going to act as a makeshift stage.

First of all, we’re going to create a new clone and store it in a variable called *noise generator*. We’ll then *tell* that clone to set the pen up, go to the center of the stage and, essentially, try to reset any possible state that could tamper with the noise generation.

Next, we’re going to wear a new random noise costume and create yet another clone, that will inherit that same costume. This new clone is going to act as our temporary stage, and thus we’re going to store it in a variable called *canvas*.

What comes next is almost exactly the algorithm we first described. We keep increasing the *noise generator* size exponentially, jumping to a random position, and stamping on the canvas. Then, we extract the costume from the canvas and store it in a variable. Only left is some

housekeeping: we delete the two temporary clones and return the noise costume.



Warping the whole process means this is going to happen between

rendering frames, so we won't even see the clones doing their thing. We'll just get a nice fractal noise costume without having affected the state of the project in any way.



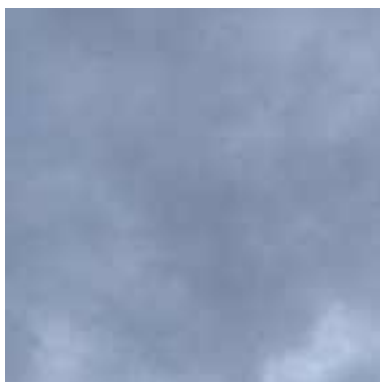
This method is shorter than the previous one and doesn't break one of the fundamental laws of functional programming, although it carries the cognitive cost of reinventing the stage as a temporary clone.

There are other ways to generate fractal noise that are closer to what you'll find on Wikipedia and academic papers, but we tend to prefer algorithms that can be *seen* as they work. In this case, we've been able to see what kind of image each step of the algorithm generates, and how they get mixed together.

In any case, now that we have our pretty noise, it's time to put it to some use.

3.2.2 Noise into Signal

Fractal noise in itself is quite pleasing to the eye, and just by playing with the amount of octaves and their transparency levels we can generate some pretty realistic textures. Images of clouds or wall paint, among others, can be further refined by just adjusting the color palette of our plain fractal noise costume, without any further computational operations.



The images on the left side are actual pictures taken by me. The top one, of a cloudy sky above my home. The bottom one, of one of my walls from up close. The ones on the right side are all generated in Snap!. They began as regular fractal noise textures, and then I used the graphic effects block to modify their color, saturation and brightness until they looked enough like the original pictures.

Given that a costume is, as we know, just a list of pixels, we could also manipulate it using hyperblocks to achieve a similar effect. For instance, we could multiply the pixels of a fractal noise image by a three element list representing a desired ratio of red, green and blue. To produce clouds, we'd leave the red channel as it was, increase the green channel by a bit, and the blue channel by a bit more.



3.3 Things that Grow

We've seen that we can color noise in ways that make it look like clouds or wall textures, but we can use brightness levels in fractal noise to represent magnitudes other than brightness itself. These values change in a smooth, seemingly organic way, and so do many other things in nature.

If we wanted to simulate natural growth, for example, we could take brightness to mean *growth direction*. We could continuously get a Snap! sprite to walk on the noise image while using the brightness value below it to determine its direction at every step.

As usual, let's begin with the most naive possible solution. We'll have two different sprites at play. One of them is just going to wear a semi-transparent fractal noise costume, the other one is going to jump around the stage and draw lines in the directions indicated by brightness levels in the other's costume.

Let's name our first sprite *Noise*, and use the  block to make it 75% transparent.

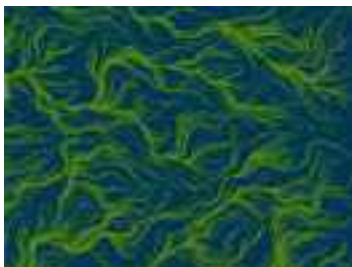


Our other sprite, aptly named *Painter*, is going to draw a bunch of 100-step long lines on stage. As said before, its direction will be dictated by the brightness levels of the sprite below, the *Noise* one.



It takes a while for the sprite to draw all these lines, but the end result is rather nice. It does bear some relationship to the original noise image, but it's almost as if we're looking at it from some other angle. This one could remind us of wood, fur or some other kind of fiber, and by tweaking the angles and pen characteristics we should be able to generate different sorts of *things that grow*.

For instance, let's scale that brightness level up by a factor of five and see what comes out of it. While we're at it, it's nice to set the transparency effect of the *Noise* sprite to 100% so it's completely out of the way. Also, in this particular case, try setting the pen color to some shade of green and the stage background to some shade of dark blue. The result should somewhat resemble floating algae.



For a texture that reminds of black marble, let's set the pen color to a very pale blue, the pen size to 0.1, the background to a very dark blue, and the brightness multiplier to 50. Let's also increase the outer loop repetitions to 2000.



If we get rid of the internal loop and move 10 steps at each iteration, we get things very much resembling a rug or natural animal fur, especially if we choose a yellowish pen color, thin pen size, and brown stage background.



And that makes sense, since individual fur strands are rather thick and don't bend so much, but they collectively describe waves and twirls.

For a striking -you'll pardon the pun- final natural phenomenon, let us choose a super thin pen size, yellow pen color and dark blue background. Now, instead of drawing needles, we'll keep drawing until we reach the edge of the stage. Since we're drawing longer lines, we can reduce the amount of iterations to just 500. We'll also apply a multiplier of 5 to the brightness, and subtract 90 from the direction so that lines sort of grow from top to the bottom of the stage. For an increased dramatic effect, we can partially bring back the noise image by making it 80% transparent. Behold the wrath of Zeus in all its glory!



3.4 Fantasy Cartography

We've just seen that, by treating it like a data source from which to extract other meanings, fractal noise has much more to offer than just cloudy textures.

For instance, we could define a threshold and divide the noise between values above and below that threshold. To do so, we're going to make use of a few clever tricks and, of course, of hyper operations:



The result very much resembles a coastline, which makes sense given the organic-looking nature of fractal noise, but how does this super short script manage to divide fractal noise into two color blobs according to a threshold?

To begin with, let's look at what the pixels reporter is all about.



Y (row)	A	B	C	D
1	110	110	110	255
2	117	117	117	255
3	120	120	120	255
4	122	122	122	255
5	123	123	123	255
6	125	125	125	255
7	126	126	126	255
8	127	127	127	255
9	128	128	128	255
10	128	128	128	255
11	128	128	128	255
12	128	128	128	255
13	128	128	128	255
14	127	127	127	255

This reporter produces a big list representing all pixels in the fractal noise image. There are 172800 rows in it, because the dimensions of our image are $360 \times 480 = 172800$.

The first three columns in this table represent the red, green and blue (RGB) color values for each pixel in the noise image. Because our fractal noise texture is a grayscale one, the three color components are equal for each pixel.

The fourth column represents the opacity of each pixel, and is mysteriously named the *alpha* channel⁴. Since the fractal noise image is fully opaque, all pixels have their alpha value maxed out at 255.

Okay, so what does applying  to this list produce?

⁴The name *alpha* comes from the mathematical formula for classic linear interpolation, where a Greek α letter represents the amount of interpolation between two images that are being mixed together. We would have called it, you know, *opacity* channel, but who are we to judge?



STUDENT	A	B	C	D
9000	True	True	True	False
9007	True	True	True	False
9009	True	True	True	False
9009	True	True	True	False
9040	True	True	True	False
9040	False	False	False	False
9042	False	False	False	False
9043	False	False	False	False
9044	True	True	True	False
9045	False	False	False	False
9046	True	True	True	False
9047	True	True	True	False
9048	True	True	True	False
9049	True	True	True	False

The  block is *hyperized* and thus can be applied to lists. In doing so, it will produce a list of booleans ( or ) that tell us whether a particular value is below 128 or not.

Now, here comes the clever trick. When applying a mathematical operation to boolean values, these will be internally converted to 1s and 0s. That is,  becomes 1, and  becomes 0:



Thus, virtually, we can think of the list above as a table of 1s and 0s.

The last trick involved in the next step is that multiplying lists of different dimensions will cap the result at the lower dimension. What this means is that multiplying a four item list by a three item list will just dump the last item from the first list and then multiply the remaining three items in order:



And, remember, booleans are understood as 1s and 0s when treated as numbers:



So, these rows where we got true values will turn into pixels representing blue⁵, while the ones with false values will turn into black. Notice that Snap! is smart about the *alpha* channel, and will just assume we mean full opacity if we take it off.

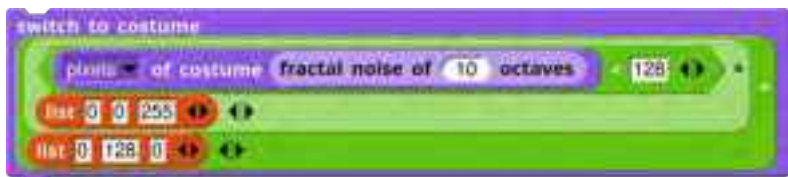
Effectively, this applies a threshold to an image. Pixels with values below 128 have become fully blue, while those with values above 128 have become fully black.

⁵In the list (0, 0, 255), the red and green components are zero, while the blue component is maxed out at 255.



Indeed, this does extract coastlines out of a fractal noise image, but it'd be nicer if land was green, instead of black.

We've just seen how black pixels are represented by a list of three zeroes, so we can maybe try to add a bit of green to the whole thing. Remember the first component of a pixel represents red, the second one represents green, and the third one represents blue.





By adding a value of 128 to the green component of every pixel, including the previously full blue ones, we've also accidentally toned down our super saturated ocean, which is just as well!

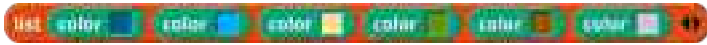
3.4.1 Fantasy Satellite Cartography

We've extracted continent and ocean blobs out of fractal noise, but in the process we've lost all the beautiful smooth gradients that the original noise had. With a little bit of extra thought, could we maybe turn this two-tone atlas-like map into something closer to a satellite image?

To get there we will need to map -pun intended- our values between 0 and 255 to a gradient of colors. Of course, the trick we used for the two-tone map will not get us there, as the number of colors is now too large. We need to think of a different strategy.

Let's begin by defining a color palette: a list of the basic colors in which we want our map to be painted. These colors should be sorted

from deepest to highest, in terms of geographic height. The first color is going to represent ocean-deep blue water, the next one will be a shallow water soft blue, moving into beach soft browns and following with forest greens, to end up in dark mountain browns and rock grays. We can make use of the  block to build this palette.



Now, this list only has six colors, whereas our fractal noise image has many more values. In particular, the noise values range from 0 to 255. If we want to map each noise value to a different color, and have these colors smoothly transition from one to another, we should stretch them into a gradient of 255 colors. This may sound like a difficult task -especially given that we want to generate smooth transitions between the colors- but a list of colors can easily be converted into an image, and luckily Snap! has us covered in the stretching images front.

The  block will take any image and re-size it to our liking. In our case, we want the resulting gradient to stay one pixel wide and grow to 255 pixels high, so we'll first wrap it into a one-element list (representing one pixel of width), then use the  block to keep the horizontal stretching to 100% and set the vertical stretching to $(255/6 * 100)\%$.



If you run this block you'll get a table with one single row, containing 255 colors describing a linear gradient that starts at ocean deep blue and ends at near white, smoothly passing through all the other colors in our original palette. Asking for the first element of that table will give us just the list of colors.

The last puzzle left to solve is how to map these 255 colors into the 255 values of the fractal noise image. Let us take a short detour to examine the exact same problem at a way smaller scale.

Imagine we want to map the names of a few animals to a list of numbers representing their indices in a list. That is, imagine we have the lists:



And we want to combine them in a way that every 1 in the second list becomes “cat”, every 2 becomes “dog”, every 3 becomes “bird” and every 4 becomes “snail”. The **Item 1 of** block should come in handy. It will give us an element of a list given its index.



As we’ve often seen, operations that work with a single number also hyper-work with a list of numbers, so just dropping the list of indices into the **Item 1 of** block will do the trick:



What we’ve just implemented is what computer scientists like to call a *lookup table*, and it’s a simple example of what we’re actually

looking for to solve the fractal noise coloring puzzle. We have a list of numbers that go from 0 to 255, and list of 255 colors that want to map it to. The list of animals is now the list of colors, and the list of numbers is now the list of fractal noise values!

Except that's not entirely true. We do *not* have a list of numbers between 0 and 255, we have a fractal noise costume, with pixel values ranging from 0 to 255. However, the list we've looking for is somewhere in the pixel values, which we can get thanks to the  block.

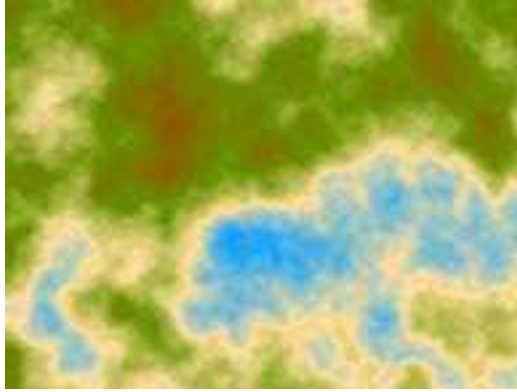


Pixel	A	B	C	D
1	108	116	115	255
2	117	117	117	255
3	120	120	120	255
4	122	122	122	255
5	125	125	125	255
6	125	125	125	255
7	126	126	126	255
8	127	127	127	255
9	128	128	128	255
10	128	128	128	255
11	128	128	128	255
12	128	128	128	255
13	128	128	128	255
14	129	129	129	255

Notice how each of the first three columns in the pixel list -representing the red, green and blue channels⁶- would serve our purpose. We can select a column in a table by using the  block in combination with .

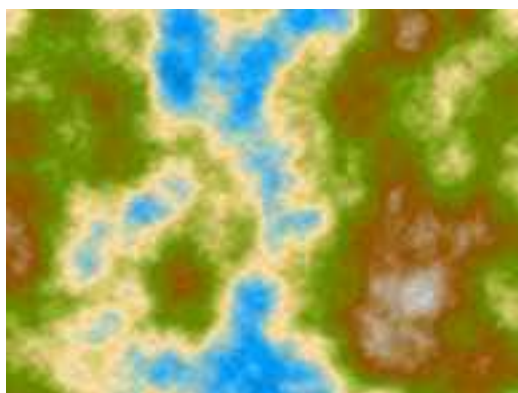


⁶The fourth column represents the *alpha* value -or opacity- for each pixel. This is a fully opaque image, so these values are all maxed out to 255.



This is now very promising, although we haven't gotten any rock gray or deep ocean blue. If we inspect the fractal noise values, we'll see that they don't *actually* go from 0 to 255. All values are within these two limits, but they never actually reach them. This is a side effect of combining all these noise octaves together and smoothing them, which has resulted in milder, not so extreme values.

We could think of several different solutions to this, but honestly the easiest one is to just add bogus values to the beginning and the end of our color gradient. Pure black at the beginning and pure white at the end will do the trick. Remember to adjust the number in the vertical scaling of the stretch block to reflect that we now have 8 colors, not 6, to distribute among 255 values.



By playing with the color palette and the number of octaves you'll find that you're able to create maps at different levels of detail for different kinds of terrain.

Who knows what the Middle Earth, Earthsea or the Land of Oz would have looked like, had their authors read this chapter in time!

3.4.2 Layered Cardboard Map

When he was young, my father used to be an avid mountaineer. His favorite spot was the Carançà valley, not only because of its breathtaking natural beauty but also because of how liberating it felt to sneak over the border through the mountains during Franco's fascist dictatorship.

So much did he like the mountains of Carançà that he spent a summer painstakingly crafting a model of the zone. By gluing several copies of a map onto cardboard, cutting them across their elevation lines, and stacking the layers on top of each other, he obtained a 3D profile of the valley that, albeit beaten and weathered, still survives nowadays.



If we took brightness values in our fractal noise image as representing elevation, maybe we could recreate my father's summer virtually, hopefully with much less work.

We'll begin by storing the pixels of a noise costume into a global variable named *elevation*. We're going to be using 9 octaves of noise in order to generate a reasonably rocky landscape, but the value can

be tweaked to your liking if you want to generate smoother or spikier terrain.



Next, we'll want to iterate over all possible brightness values, between 0 and 255, and stamp a costume that represents a cutout of the noise image for that particular value.

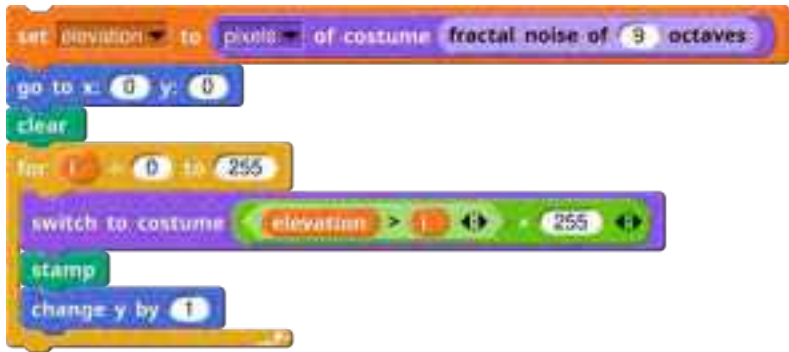
Our first approach is going to be to hyper-check whether all pixels of the noise image are above the current threshold value. That's going to give us a list where each brightness value in the noise image has been replaced by either  or , depending on whether that particular brightness value is above the current threshold.

To see the kind of data we're dealing with, let's run the hyper-check against one particular threshold, like 128.



To turn this list of  /  values into a list of numbers, we can just multiply it by a number. We'll get that number in place of

`true` , and 0 otherwise. Let's try to multiply it by, for instance, 255.



After a rather long wait, an image resembling a CT scan is going to start forming on the stage. However, as can be seen in the sequence of images above, every subsequent costume is being stamped on top of the previous one, completely covering it. If images weren't completely opaque, they'd probably mix together much better.

Remember the fourth dimension of color in the pixel list: the alpha channel? So far, we've been ignoring it, but this time we're going to make use of it to make the intermediate images semitransparent.

Right now, the resulting costumes look like this:

172800	A	B	C	D
1911	0	0	0	255
1912	0	0	0	255
1913	0	0	0	255
1914	0	0	0	255
1915	0	0	0	255
1916	0	0	0	255
1917	0	0	0	255
1918	0	0	0	255
1919	0	0	0	255
1920	0	0	0	255
1921	255	255	255	255
1922	255	255	255	255
1923	255	255	255	255
1924	255	255	255	255

Notice how the fourth column, representing opacity, is always maxed out at 255, no matter the value of the other columns.

Essentially, we want to have this exact same costume, but with the fourth column multiplied by a low factor so it mixes with the subsequent

ones when it is stamped.

We could try exactly that: multiplying the costume by a list of four numbers. The three first components we want to leave as they are, so we'll multiply them by one. The fourth one we'll want to make very small, so we'll multiply it by a small factor, like 0.01.

This is going to be even slower than molasses, but do not despair. We're following Kent Beck⁷'s mantra of: *«make it work, make it right, make it fast»*, and we're currently at the making it work phase.



If you've been patient enough, you should have gotten something like the image above on the Snap! stage. Now that we got it working, let's make it right.

Instead of making each layer semitransparent, why not make the parts outside of the threshold fully transparent and leave the rest of the texture the way it was? To do so, we'll want to construct images where the first three columns are left untouched (that is, they're the ones in the elevation list) while the fourth column is zero for values below the threshold, and 255 for values above it.

⁷This statement has been attributed to several people. We're choosing Kent because he's a smalltalker, but it may be the case that someone else, like Stephen C. Johnson and Brian W. Kernighan, wrote it first.

This will require a bit of matrix gymnastics. To begin with, we'll want to start with just one of the meaningful columns in the noise texture. For instance, the first one. The elevation variable assignment should thus look like this:



Next, inside of our loop, we'll want a costume where the three first columns are identical to the ones in the *elevation* variable, while the fourth one, the opacity, is zero for values below the threshold and 255 for values above it. Something like this:



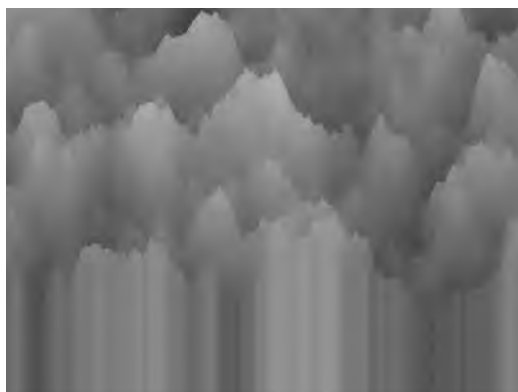
The only problem with this list is it's rotated 90° from what Snap! expects as a costume:

,VVVVVVV\WWWWWXXXXXXX\YYYYYYY\ZZZZZZZ\AAAAAAA\BBBBBBB\						
127	128	128	128	129	130	132
127	128	128	128	129	130	132
127	128	128	128	129	130	132
0	0	0	0	255	255	255

But that's easily solved by applying `columns of` to the table, the resulting script becoming:



This generates a better image than before, and it does so significantly faster.



There's still room for improvement, though.

Notice how the first few tenths of values seem to never hit the threshold, as do the last ones, so it'd be nice to not compute those

at all. We can achieve that by iterating between the minimum and maximum values in the elevation list. This should shave off roughly half the processing time.

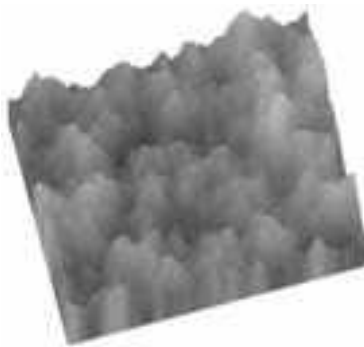
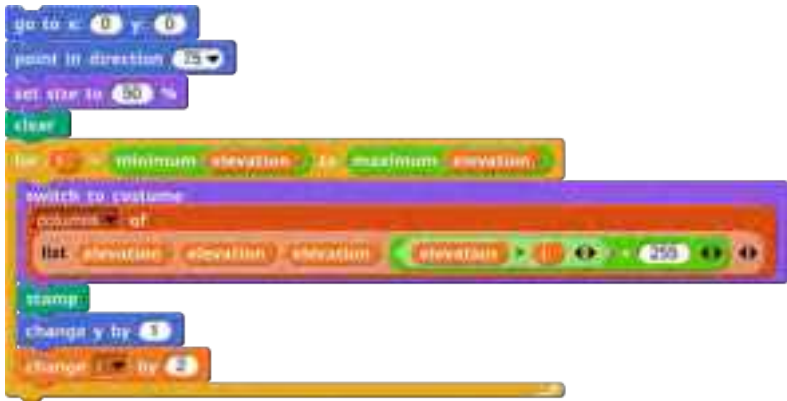
To get the minimum and maximum values of a list, we can drop said list into the *expansion arrows* of the  and  blocks. When dragging the list over the  symbol you'll notice an orange halo popping up indicating that this is a valid drop spot.



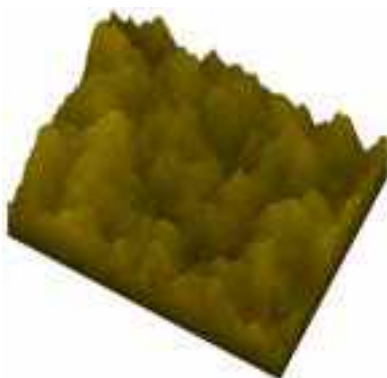
We're also probably taking way too many measurements. The elevation lines in this map are extremely crammed together. By skipping two iterations at every step we'll save about one third of the time, while only incurring in a slight loss of vertical resolution and making the 3D model flatter.



To make the 3D effect more prominent, we can also scale the whole thing down to 50% and rotate it a few degrees. The final script and result looks like this:



Wouldn't it be pretty if we could turn this into a rotating animation, instead of just a static rendering? Luckily, we just need a few minor changes to achieve the effect. Instead of stamping these slices, we'll make clones that wear these costumes. When we're done with creating all the clones, we'll tell them all to rotate continuously. While we're at it, we can also tweak the three color components to make them look more earth-like, even making use of the increasingly high values of i to make the peaks brighter than the valleys.



3.5 Why is it Awesome?

In the age of generative artificial intelligence, it may seem pointless to use old algorithms to generate images like the ones we've been making using fractal noise. However, on second thought, we've been generating stunning textures and credible maps with virtually no computing power at all. State-of-the-art GenAI tools generate not only media, but also colossal amounts of CO₂, while spending as much energy and water as entire countries.

Ethical and ecological considerations aside, the very simplicity of the algorithms presented in this chapter and the fact that they very quickly produce realistic textures out of chaos is quite awesome. The idea of modeling the kind of randomness we find in nature and obtaining natural-looking results from it is exciting and very rewarding.

Perlin noise is far from obsolete. It is still widely used in game development today, along with algorithms that emerged from it, such as Simplex noise, Worley noise, and all sorts of different kinds of fractal noises. These algorithms are so small and fast that they can be used to generate textures, maps, levels or 3D models in real time with practically no computing cost. Most game engines come with primitives that abstract away the algorithms and produce ready-to-use noise. Another advantage of many of these noise algorithms is that they can be instrumented to produce repeatable and infinitely tiling seamless textures, making them useful to generate credible terrains and game levels in real time while making sure that different areas match together perfectly.

All in all, Perlin is still proudly living up to the Oscar it got in 1997, and will most certainly still be relevant many decades from today.