Conor King

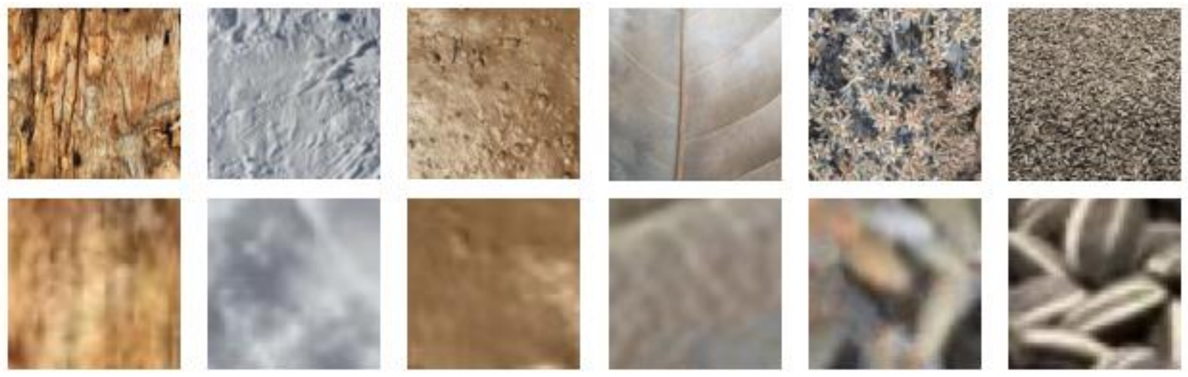
HW 2: Texture Synthesis by Non-Parametric Sampling

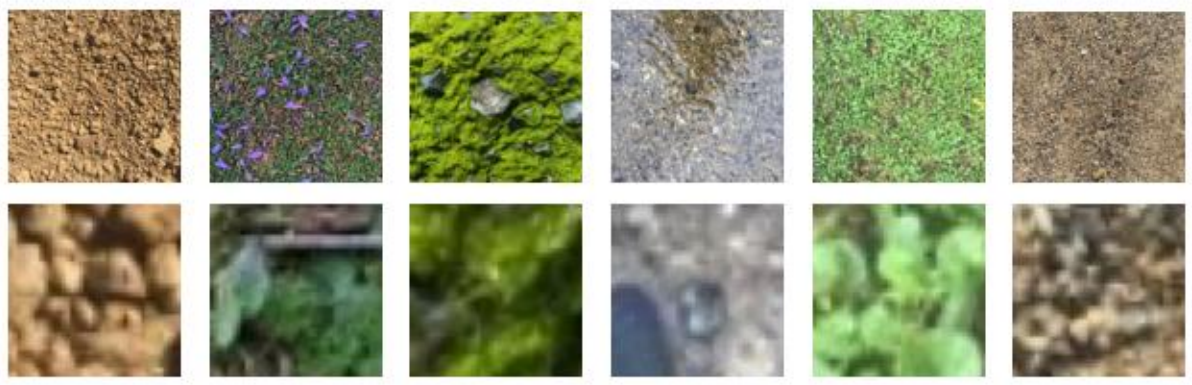
Speed Up

My initial attempt used for loops to compute the SSD between the target patch and every patch of the original texture. However, I found that when this approach was used on the provided textures, generating a single pixel took approximately two minutes. That was not a viable rate for producing a synthesized image of reasonable dimensions. My laptop does not have a GPU which supports GPU-acceleration (at least using pytorch and CUDA) and I did not observe a speedup when I attempted to use CUDA on Google Collab. I required a more efficient technique.

I found a straightforward implementation of the original paper in a github repository. This implementation uses numpy's stride technique to efficiently compute all SSDs simultaneously. I built my project based on this repository.

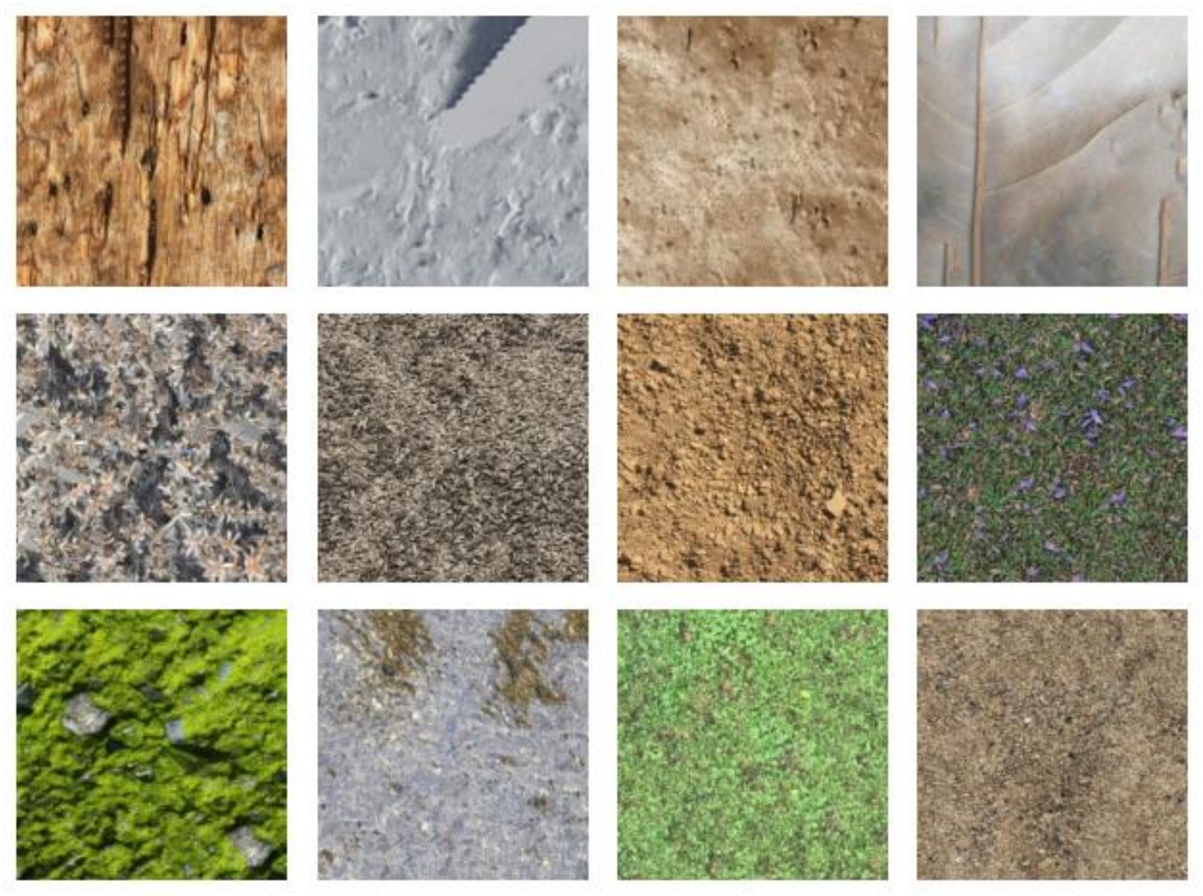
For window sizes of 11, 31, and 51 pixels, I synthesized new textures from the 12 provided textures for 15 minutes each. The results for the window size of 11 are provided below.





15 minutes produced very small images for the 11 pixel windows; the 31 and 51 pixel kernels were too small to be recognizable. To be really effective, an even faster method was necessary.

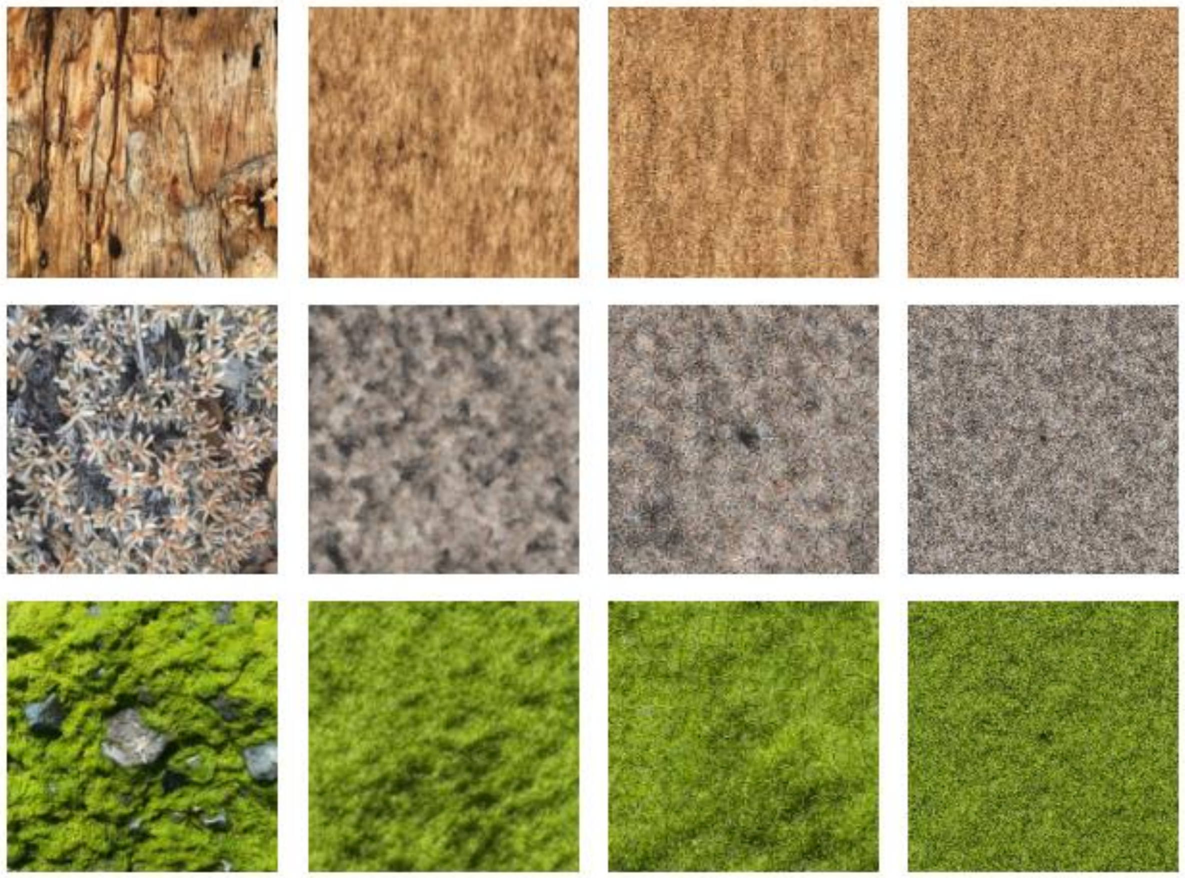
As the size of the original texture increases, the time taken to produce a pixel increases exponentially. The most effective way to speed up the process would be to reduce the size of the original texture. I resized each texture to a 64x64 pixel image by reducing the resolution. Rerunning the program on these new images with window size of 11 for 25 minutes each produced the following results.



These are much larger that the previous method and on the whole look fairly good compared to the originals. There are obvious flaws for 2 (snow) and 4 (leaf), and poor quality areas in 6 (sunflower seeds).

The downside of shrinking the original textures by simply decreasing the resolution is that the resolution of the synthesized textures is likewise decreased. To combat this problem, I considered this: since the texture is stochastic, any given area of the texture should be statistically equivalent to any other section. Therefore, rather than reducing the size at the beginning, it should be possible to simply select a random 64x64 patch of the original texture whenever the SSDs are to be calculated. The results of this approach are shown below, for 3 of the original images and for window sizes 11, 31, and 51.

Conor King



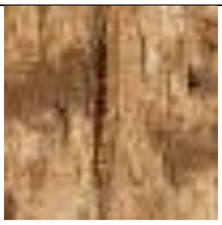
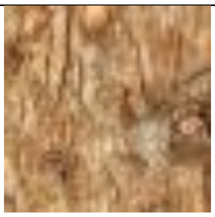
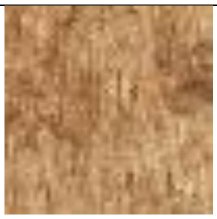
The results are significantly more blurred than the previous method; this may be due to the window sizes being appropriate for a reduced image, but too small for a patch of the full-sized image.

Randomization

One issue with the repository's method, and with my method up till this point, was this: the threshold for acceptable SSDs was calculated by multiplying the minimum SSD by a small amount. In principle this works well; however, as the process begins by growing outward from a 3x3 patch of the original texture, the first pixel has a neighborhood which is an exact match with at least one area of the original, meaning an SSD of 0. Therefore the threshold was 0 and only exact matches were used to select the pixel. This meant that the same pixel was used as was in the analogous location in the original texture, and so the problem was perpetuated. The only way it could end and begin generating novel patterns was by reaching the edge of the original texture.

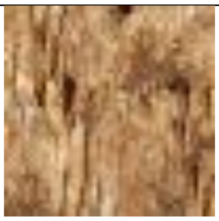
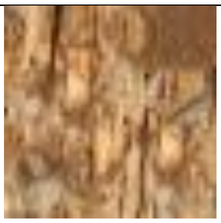
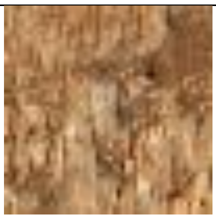
To rectify this, I used two methods. The first was simply selecting from among the N lowest SSDs each time, rather that using the lowest SSD. The results for N=4 and N=20 applied to the first texture are shown below.

Conor King

	N = 4	N = 20
Uniform		
Weighted by SSD		

It is difficult to verify by inspection whether the textures are more original, but they do seem to be. The N = 4 images are less blurred, more similar to the original texture, which is reasonable, as only better matches are used. I cannot detect an appreciable difference between uniform and weighted selection.

The second method I tested was finding a large number of sample SSDs between random patches of the original texture in order to establish which SSD would establish a certain threshold.

	Threshold = 0.003	Threshold = 0.0003
Uniform		
Weighted by SSD		-

I cannot detect a appreciable difference between any of these images. When the threshold = 0.0003, only a single patch was generally allowed, so choice of selection process is immaterial.

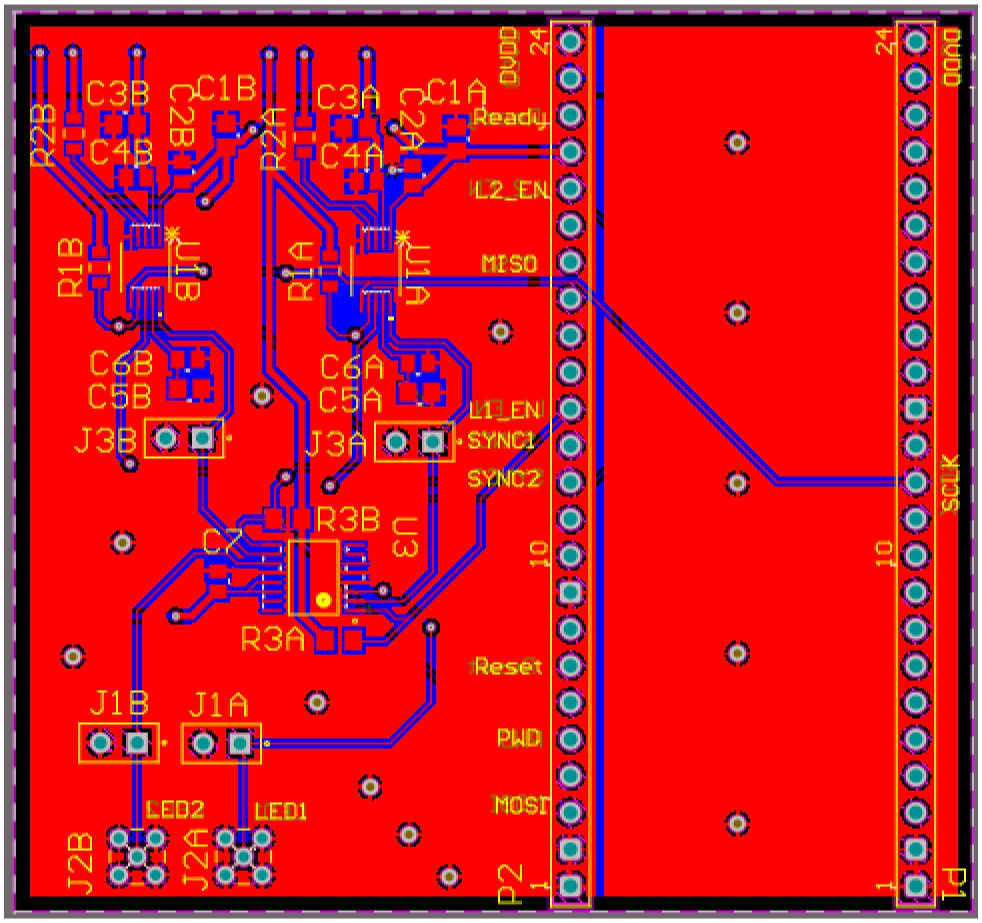
Additional Images

I did not use the MNIST data; the enormous number of patches produced would have stalled out the process. However, I did extend the method to some textures of my own.

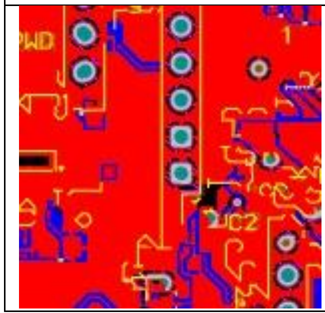
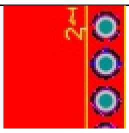
Firstly, a PCB schematic I designed.

Original:

Conor King



Synthetic

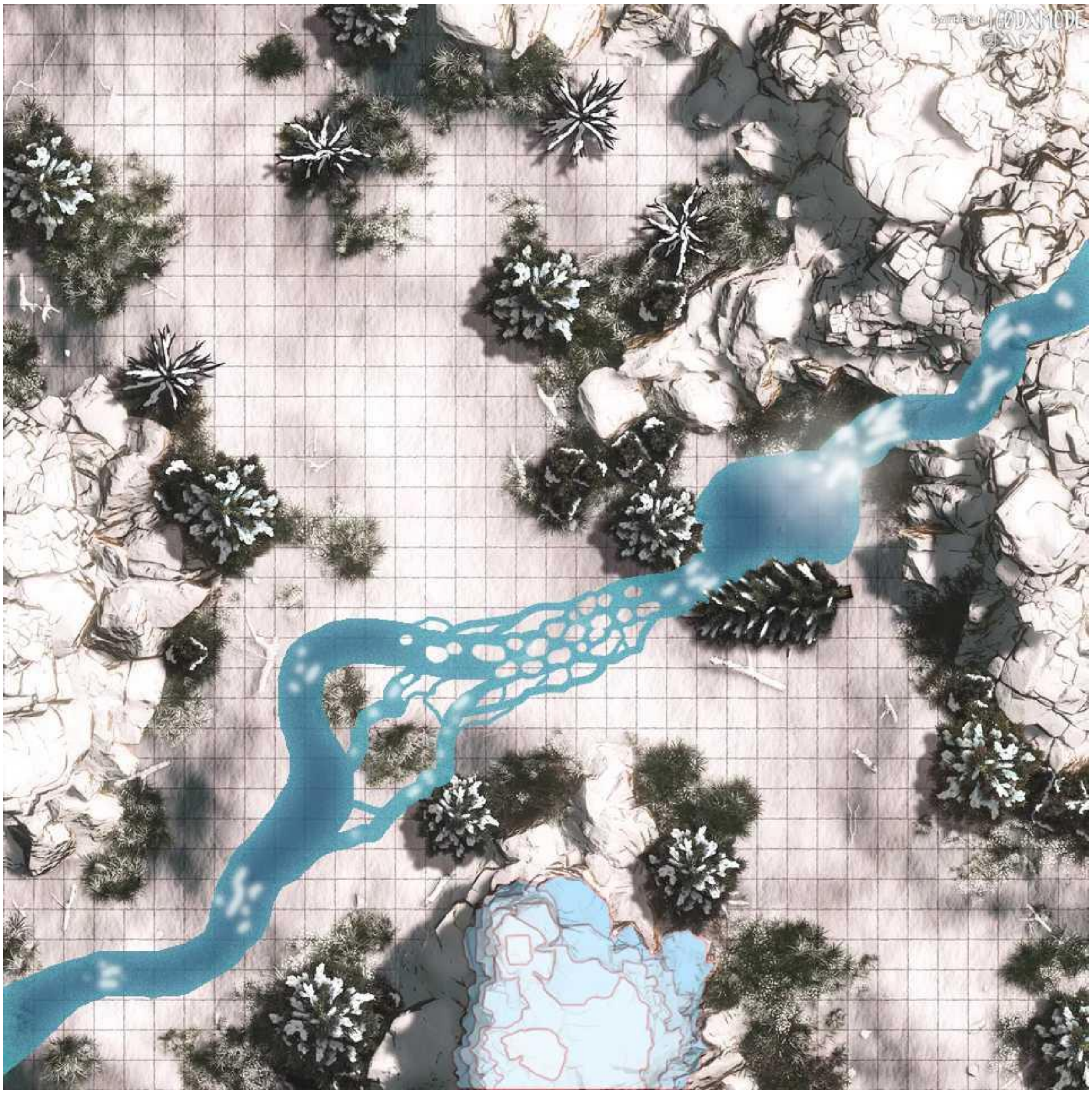
Window = 11	Window = 31	Window = 51
		

As the window size increased, the closeness to the original appears to increase, although the slow speed and consequently small size make this difficult to establish.

Conor King

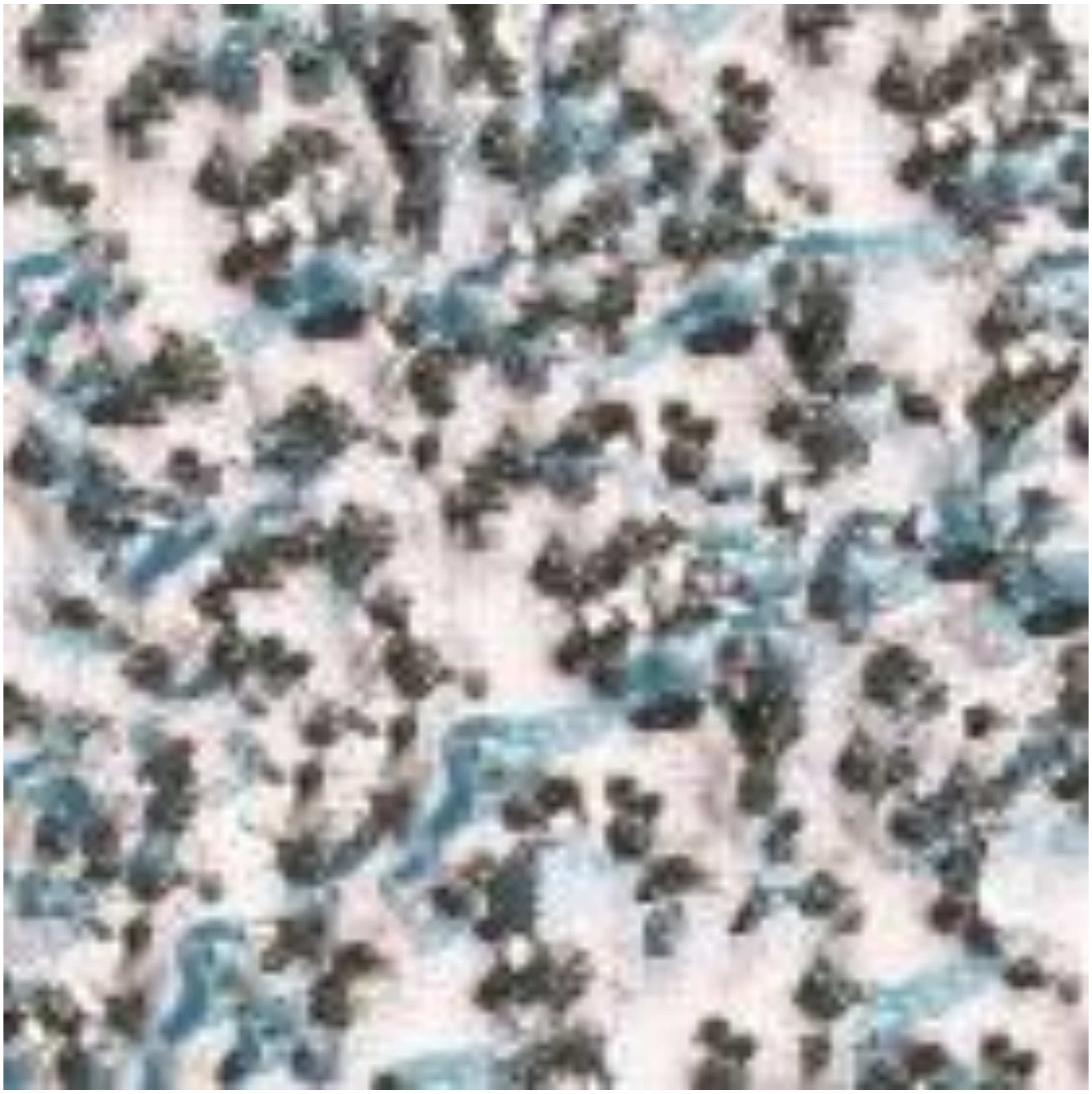
Secondly, to a battle map from my Dungeons and Dragons campaign. I was interested to see how the method would handle a grid combined with a more natural stochastic texture.

Original:



Conor King

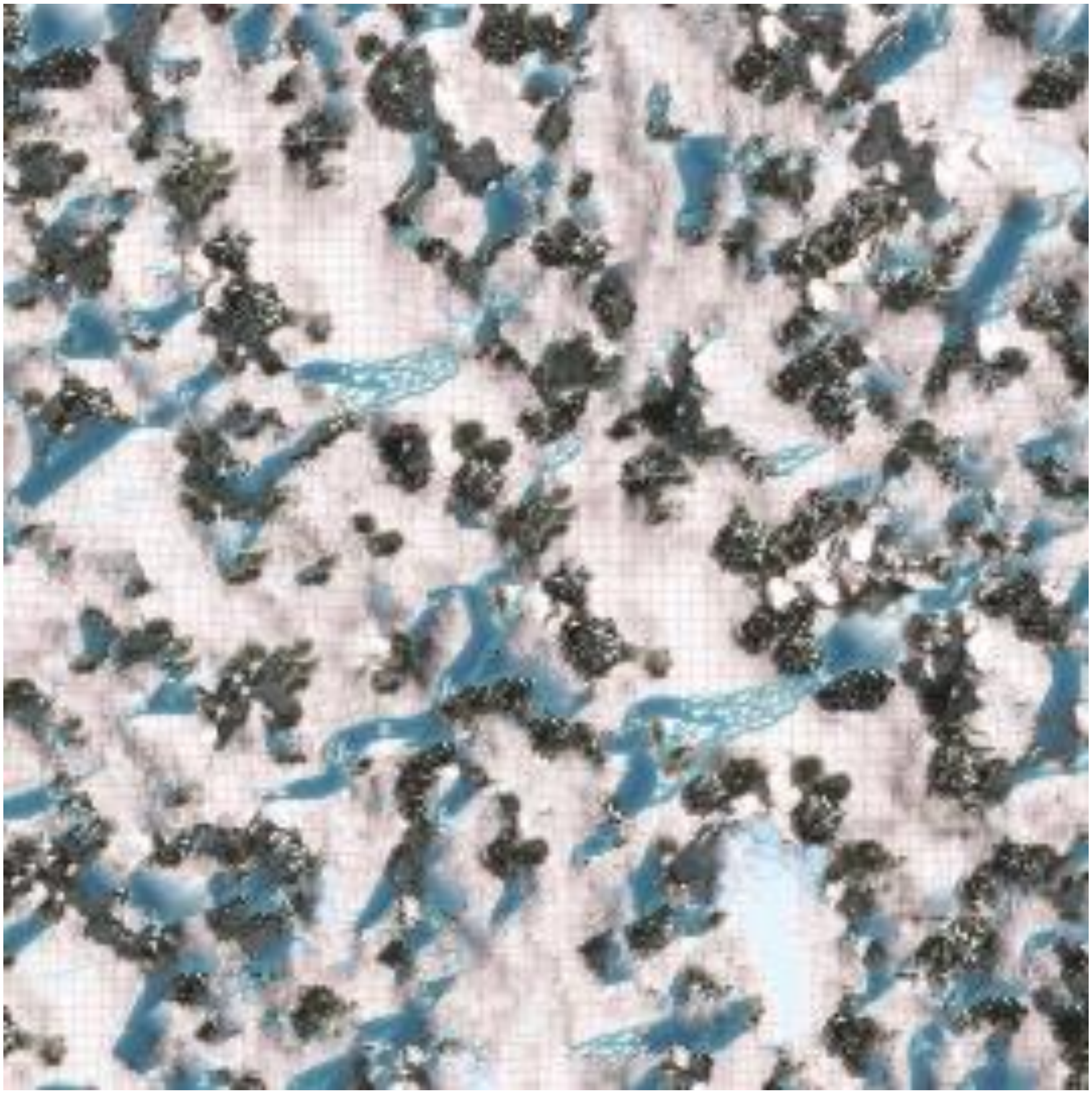
Reduced to 64x64 with a window size of 11, running for 10 minutes:



Much lower resolution of course, thought the grid can still be seen.

Conor King

Reduced to 64x64 with a window size of 31, running for 1 hour



Still poor resolution, but the larger window size produces spacing more similar to the original.

Full Size with a window size of 11 running for 15 hours

Conor King

This demonstrates a) the extremely slow synthesis rate when the original texture is large and b) the exact copying mentioned in above. This image is reproduced from the original exactly.

128x128 with a window size of 31 running for ~6 hours.