Procedural Generation of Brick, Concrete, and Tiles as Textures

Chris Chen & Owen Elliff

29 April 2019

Abstract

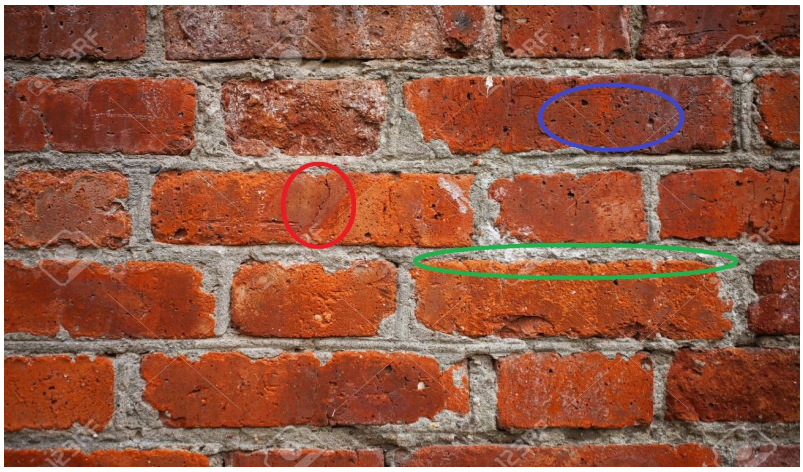
In this paper, we describe the techniques used by others in the field of procedural texture generation as concerning brick, concrete, and tile textures. We also follow this with description of our implementation of procedural texture generation of said types of textures, as well as description of our attempt at implementing some of the aforementioned techniques.

1 Introduction

As it stands, there is no single reliable method to procedurally produce textures for brick, concrete, and tiles such that they can cover all variants of brick, concrete, and tiles as they exist. Our goal was to come up with several different methods to produce non-photorealistic textures of brick, concrete, and tiles. In the case of bricks, the most time and energy was put forth, as it was found that many concrete and tile textures are simpler than brick textures with a lower variance in types of features that need to be reproduced.

Of the many features that occur in bricks and brickwork, there are several more notable ones that were our points of focus. In brickwork, bricks are often placed in a staggered pattern, their most signature feature from afar. Looking at them from up close, bricks themselves are speckled with very small, surface level holes. As well as this, the passage of times results in two other distinct features; erosion of the edges of each brick, and short cracks that rarely span a whole brick.

A crack is circled in red, erosion is circled in green, and holes are circled in blue.



We took three different approaches to these features. For the brickwork pattern, we focused on a simple data representation, and later a more complex one. In the case of the various features found in bricks, we used noise and simple stochastic processes. Finally, in the case of the overall texture and color of the bricks, we used simple image processing of bricks and other images.

2 Background

There is much prior work that our methods build upon.

J. Legakis et al, 2001 introduces a method to generate 3D texturing that allows for multiple kinds of patterns to be created and applied, by use of occupancy maps and pattern generators. An occupancy map is a map of which cells are already occupied by their neighbors and which cells still need to be textured, and pattern generators are higher-level structures that describe how an object's texture should be generated based on if that part of the object is a corner, an edge, a face or based on it's label. This allows for 3D texture to be generated for a model that have consistent bricks between all faces, along with variable sized blocks and other ways to generate textures based on the given pattern generator. We wished to implement this paper after generating our base structure due to the paper's relative complexity, but were unable to given the time constraints.

Perlin, 1985 and Perlin, 2002 both introduced methods for producing believably random noise by producing a field of gradient vectors so that any point has a specific value determined by the direction of the nearest vectors. This is regularly used in two and three dimensions to give procedurally produced textures and geometry that have a natural appearance to them. As it concerns this paper, these forms of noise were both used in the cases of coloring the textures and adding small holes to bricks and concrete.

Also important to this paper is the mathematical concept of the random walk. The random walk is a simple process by which given a starting point in any dimension, a theoretical person may take a step in any direction. Over a long period of time, these steps will produce an effect similar to that found in nature. We used this to simulate the effects of erosion and cracking in our bricks.

Finally, A. Lagae et al, 2010 focused on processing images to reproduce the patterns found within them. It did this by treating each image as a combination of different wavelengths of wavelet noise (R. Cook et al, 2005). They applied Fourier transforms to the images to obtain the different weights for the wavelengths of wavelet noise that might exist in the image. This method worked very well for images of grass and marble, but not so well for images with structure to them. We hoped to use this method to achieve noise for individual bricks, concrete, and tiles with mixed success.

3 Data Structures

To represent each brick or tile, we store the center of each tile as a two dimensional vector. The points are generated intuitively and handle a number of bricks lengthwise and widthwise. If the brick or tiles are staggered, the center points at the appropriate rows are offset as well. The width and length of each brick are generated mathematically as well. This as well is intuitive to generate. Each brick's edge erosion is stored in a list of ints, and the method that they are generated is by using random walk at the edges of each brick.

To generate the texture, we use two passthroughs, the first passthrough being rather simple and the second passthrough being somewhat more complex. The first passthrough is to add the mortar color to the entire texture, also using Perlin Noise to make it look less repetitive. The second passthrough is going to each of the stored points. At each of those points, we generate either brick, tile, or concrete textures. Then we set the texture's color at each of those points equal to the previously generated colors starting from the upper-right corner of each brick and ending at the lower-left corner. If the option to use the generated image noise is selected, we use that generated image noise instead of generating a texture for brick, tile, or concrete. However, if the point would result in changing a color outside of the generated eroded edge, we instead ignore that point. We simply compare the checked point's x or y coordinate related to each point's eroded edge at it's related coordinate.

The reason for these two passthroughs is twofold. First of all, it works rather well with modifying bricks. If we wish to move the points to relocate the bricks and simulate them in a different structure or manipulate them in another manner, we do not need to modify the mortar as well. The second reason is that this structure is rather intuitive to understand. Each brick is represented easily, and the manner that they are rendered upon the texture is also simple.

Another possible implementation that could have been done is the implementation presented in J. Legakis et al, 2001. This implementation was attempted after the previous structures we created to expand upon them, but lack of time resulted in an incomplete implementation.

4 Reproducing Features in Bricks, Extension to Concrete and Tiles

We worked primarily in a total texture resolution of 1024 by 1024 pixels. The values listed below are dependant upon this and if the texture is scaled, the values should be scaled in the same manner for the x and y of the new resolution respectively. As well, with all values listed below, they can of course be modified as needed, but we chose these values as we found them to produce our desired results.

To begin with, we initially focused on producing the color variance found in bricks. For the color variance, we used 2D Perlin noise. We gave the noise the x and y values of each pixel on each brick, multiplied those x and y values by values ranging from 0.01 to 0.004, finding 0.006 to be our preferred value, and then added a large, random offset to those now multiplied x and y values. We then multiplied the returned noise value by a chosen red-orange color. This resulted in each brick having a distinctive shade of color, as well as some variance in color over itself. This also resulted in each brick not looking like they were part of one, larger brick that had been cut into smaller bricks, as this is not an effect found in brick walls.

To simulate the small holes found in bricks, we used Perlin noise in the same way as before, but instead multiplying the values by 0.1. The noise result was then cut off at 0.05 and set to 1 if it was above that number. If below 0.05, the noise was then multiplied by 1 / 0.05 to achieve a smoothness from the holes center. The obtained values for the holes were then used either to multiply against the color, or to generate normals, or both.

Next, to generate the cracks, we pick a random point on the brick's edge. We then choose a direction to start as the normal against that edge. This direction serves as the starting point for a random walk, where the walk is in a set increment of 1.5 and the direction is modified by .1 in the x or y every step, coming with a 5% chance at any step for the crack to cease. The crack also

stops on reaching another edge. Giving the crack a small radius, we can once again either multiply the color value by zero where there is a crack, or create normals from the cracks.

Finally, to generate each of the brick's erosion at the edges, we apply random walk at each of the brick's edges. At each pixel, the random walk will either move one unit upwards or inwards, not move, or move one unit downwards or backwards. Each of these moves are based on the previous random walk position. The generated positions also tend not erode based on how much the brick was previously eroded. For example if we are generating erosion at point (1, 4) in the y direction, the higher the y value is from 4 the less likely it is to increase. We also introduce a max erosion height to prevent extreme cases.

5 Image Processing

Our attempt to implement the image processing from A. Lagae et al, 2010 was unsuccessful from one standpoint. However, in attempting to implement something similar to their method, we came up with a simple solution to determine the wavelength of noise in any given image. The solution goes as follows:

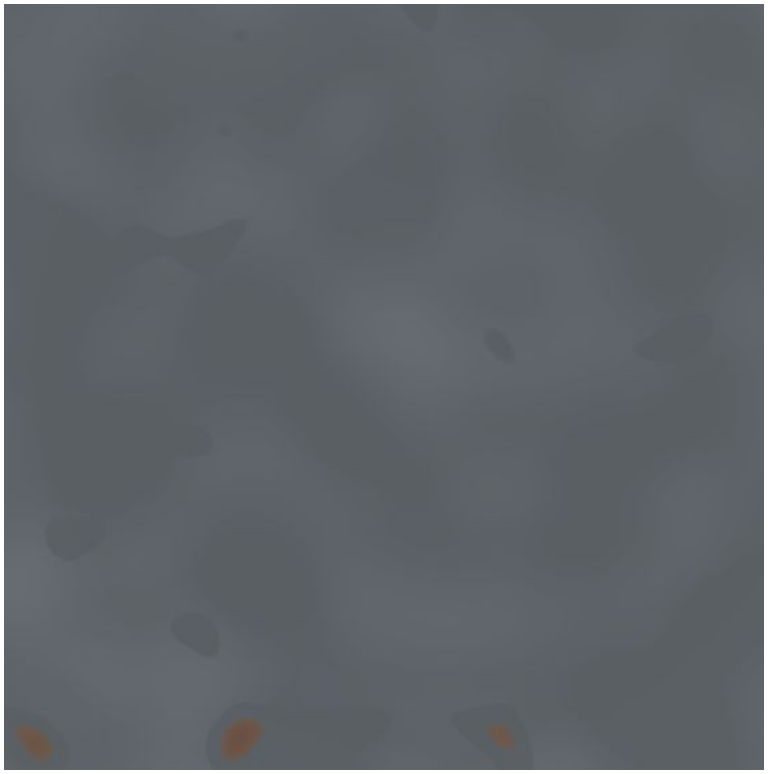
```
float v = 0
        for each row i totalling in n rows
                for each pixel j in row of pixels of length m
                        d = difference between this and last pixel in color value
                        d = d * (0.5)
                        v += d
                v = v * (0.5)
        repeat for each column
w = add v's together and divide it by (n * m * 0.2)
w is your wavelength
```

The above solution manages to reliably find the wavelength of the Perlin and Simplex noise in an image with the largest error we found being a 20% deviation from the correct wavelength. We assume this not to be a new solution for this problem, but we came up with it ourselves and have a poor understanding of why it works.

In order to achieve a more complex pattern for our bricks using this resulting wavelength, we added two higher wavelengths of noise to the texture which gave a better looking brick.

6 Results

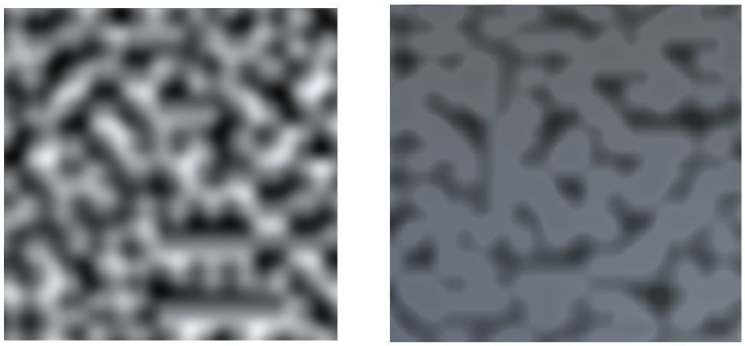
This is an example of other extensions we can do, such as a metal texture.



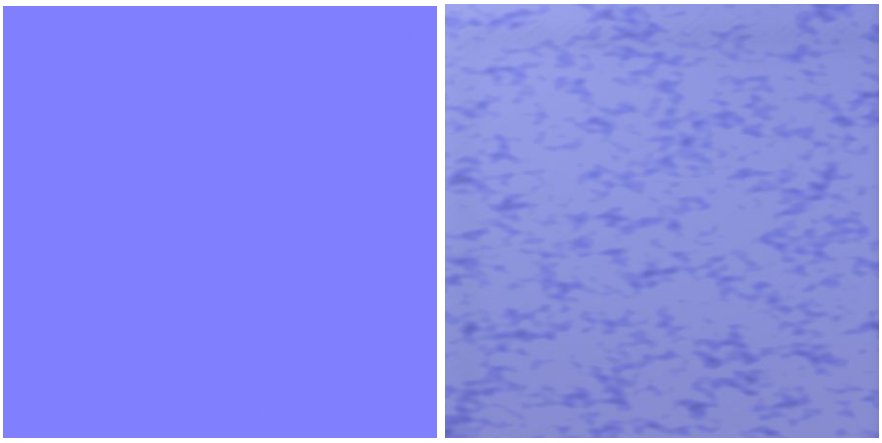
Here is a picture of a complete brick texture, with correct normals. We do not use the image synthesizer in this texture.

An example of our brick texture applied to a 3D model

Image processing with input on the left, and the output on the right



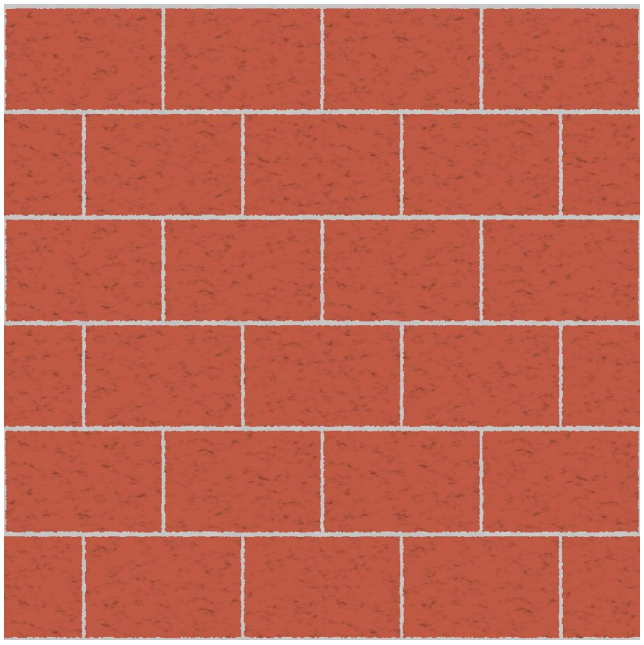
A texture generated from feeding an image of normals into the image synthesizer.



A simple extension of the image processing to result in a potato.

An example of a generated brick texture, using our image synthesizer.
The given image, several bricks stitched together to create one, larger brick.

The resulting image.



An example of incorrect normals

An example of a fail case for the image processing

7 Conclusions

In conclusion, we managed to come up with two different solutions for creating believable, non-photorealistic bricks, concrete, and tiles with noise and stochastic processes, as well as one solution for representing the bricks as data. While there certainly are flaws in our

implementations, they do manage to produce results that are believable as bricks, concrete, and tiles.

The work we've done here could certainly be used to produce more textures than those which are shown, and the simple image processing could easily be used to quickly find the desired overall wavelength of an image instead of needing to fiddle with the noise values manually as we were forced to do for most of this project. As well, the data structure allows for brick placements in ways that are more than just the simple tiling that we have. A possible expansion on that could be to generate bricks in a circular formation or position them algorithmically. There is a lot that this can be expanded upon in the future.

8 Citations

Lagae, A., Vangorp, P., Lenaerts, T., & Dutre, P. (2010). Procedural isotropic stochastic textures by example. *Computers & Graphics,34*(4), 312-321. Retrieved April 2, 2019, from https://www.sciencedirect.com/science/article/pii/S0097849310000713?via=ihub#!

Lefebvre, L., & Poulin, P. (2000). Analysis and Synthesis of Structural Textures. *Graphics Interface 2000*. Retrieved April 2, 2019, from https://www.semanticscholar.org/paper/Analysis-and-Synthesis-of-Structural-Textures-Lefebvre-Poulin/1015375e9b1b21f6d0669c148cc9b3790e6fcf9e.

Perlin, K. (1985). An image synthesizer. *ACM SIGGRAPH Computer Graphics,19*(3), 287-296. doi:10.1145/325165.325247

Perlin, K. (2002). Improving noise. *ACM Transactions on Graphics,21*(3). doi:10.1145/566654.566636