A Cellular Texture Basis Function

Steven Worley[1]

ABSTRACT

Solid texturing is a powerful way to add detail to the surface of rendered objects. Perlin's "noise" is a 3D basis function used in some of the most dramatic and useful surface texture algorithms. We present a new basis function which complements Perlin noise, based on a partitioning of space into a random array of cells. We have used this new basis function to produce textured surfaces resembling flagstone-like tiled areas, organic crusty skin, crumpled paper, ice, rock, mountain ranges, and craters. The new basis function can be computed efficiently without the need for precalculation or table storage.

INTRODUCTION

Procedural texturing has proved itself valuable in image synthesis, allowing for complex surfaces to be rendered without requiring image mapping or hand modeling of geometric details. The most useful texturing techniques were introduced by Perlin [7] with his introduction of a fractal noise basis function which has become the primary tool used in procedural texturing. Since the fractal noise basis does not need any storage or precomputation, and returns a value for all locations in $\Re^3$, it is easy to use and applicable to many applications. The scalar value can be directly mapped into a color, but it is often used for spatially perturbing regular patterns. Its derivatives can be used for bump mapping. [1, 3, 5, 6, 7, 8]

Many texture algorithms are not as broadly applicable as noise because of their limited form. For example, the class of *reaction-diffusion* textures, as in [10], provides interesting surfaces, with a character that provides features such as spots and stripes. An algorithm by Miyata [4] generates impressive stone wall patterns. These methods, while powerful, aren't as generally useful as noise since they require extensive precomputation and don't return a scalar value to be used as part of a larger texture.

The simple functional nature of noise makes it an adaptable tool one might call a texture "basis" function. A basis function should be a scalar value, defined over $\Re^3$. This allows it to be used in the same manner noise is applied. Fourier methods [1, 2] produce effective basis functions, but usually similar appearances can be produced more easily using noise. The immense utility of the noise function motivates us to find new texture functions that can also be used as basis functions, so they may be used in the same versatile manner that noise is used.

[1] 405 El Camino Real Suite 121, Menlo Park CA 94025 E-mail: steve@worley.com

In this paper, we propose a new set of related texture basis functions. They are based on scattering "feature points" throughout $\Re^3$ and building a scalar function based on the distribution of the local points. The use of distributed points in space for texturing is not new; "bombing" is a technique which places geometric features such as spheres throughout space, which generates patterns on surfaces that cut through the volume of these features, forming polkadots, for example. [9, 5] This technique is not a basis function, and is significantly less useful than noise. Lewis also used points scattered throughout space for texturing. His method forms a basis function, but it is better described as an alternative method of generating a noise basis than a new basis function with a different appearance.[3]

In this paper, we introduce a new texture basis function that has interesting behavior, and can be evaluated efficiently without any precomputation. After defining the function and its implementation, we show some applications demonstrating its utility.

nTH-CLOSEST POINT BASIS FUNCTION

We can define a new basis function based on the idea of random *feature points.* Imagine points randomly distributed through all of $\Re^3$. For any location $\mathbf{x}$, there is some feature point which lies closer to $\mathbf{x}$ than any other feature point. Define $F_1(\mathbf{x})$ as the distance from $\mathbf{x}$ to that closest feature point. As $\mathbf{x}$ is varied, F_1 varies smoothly since the distance between the sample location and the fixed feature point varies smoothly. However, at certain cusp locations, the point $\mathbf{x}$ will be equidistant between two feature points. Here, the value of $F_1(\mathbf{x})$ is still well defined, since the value is the same no matter which feature point is chosen to calculate the distance. Varying the position of $\mathbf{x}$ will return values of F_1 that are still continuous, though the derivative of F_1 will change discontinuously as the distance calculation "switches" from one feature point to its neighbor.

It can be seen that the locations where the function F_1 "switches" from one feature point to the next (where its derivative is discontinuous) are along the equidistance planes that separate two points in $\Re^3$. These planes are exactly the planes that are computed by a Voronoi diagram, which by definition partitions space into cellular regions where all the points within each region are closer to its defining point than any other point.

The function $F_2(\mathbf{x})$ can be defined as the distance between the location $\mathbf{x}$ and the feature point which is the *second* closest to the $\mathbf{x}$. With similar arguments as before, F_2 is continuous everywhere, but its derivative is not at those locations where the second-closest point swaps with either the first-closest or third-closest. Similarly, we can define $F_n(\mathbf{x})$ as the distance between $\mathbf{x}$ and the nth closest feature point.

The functions F have some interesting properties. F_n is always continuous. F_n are nondecreasing; $0 \leq F_1(\mathbf{x}) \leq F_2(\mathbf{x}) \leq F_3(\mathbf{x})$. In general, $F_n(\mathbf{x}) \leq F_{n+1}(\mathbf{x})$ by the definition of F_n. The gradient of F_n is simply the unit direction vector from the nth closest feature point to $\mathbf{x}$.

COMPUTATION OF $F_n(\mathbf{x})$

To evaluate the functions F_n, we must first define how feature points are spread through space. The density and distribution of points will change the character of the basis functions. We want an isotropic distribution of points to avoid artifacts such as an obvious grid-like pattern. This eliminates any regular lattices such as a cubic spacing. Even if the lattice points are jittered, the underlying lattice structure may cause odd patterns.

The simplest distribution is simply a Poisson distribution, which specifies a mean density of points in space, with the location of each point being independent of the other points. The expected number of points in any region is simply the point density times the volume of the region. There may be more or less than this expected number of points in the region; the exact probabilities of any number of points in a region can be computed by using the discrete Poisson distribution function.

Our approach divides space into a grid of uniformly spaced cubes, separated at the integer coordinate locations. Each "cube" in space can be uniquely represented by its integer coordinates, and by simple floor operations we can determine, for example, that a point like (1.2, 3.33, 2.3) lies within the cube indexed by (1, 3, 2).

Each cube in space may contain zero, one, or more feature points. We determine this on-the-fly quite simply by noting that the Poisson random distribution function describes the exact probabilities of each of the possible number of feature points occuring in the cube. For a mean density of λ feature points per unit volume , the probability of m points occuring in a unit cube is $(\lambda^{-m}\, e^{\lambda}\, m!)^{-1}$. Thus we can tabulate the probabilities for $m = 0, 1, 2, 3...$ and index a random number into this table to determine how many feature points fall within this cube. In practice, we use a density of about $m = 4$, but clamp the points in each cell to range between 1 and 9, for efficiency reasons discussed later. This cutoff in theory breaks some of the isotropy of the distribution of feature points, but in practice we see no visual consequence.

The "random" number we use to determine the number of feature points in a cube obviously must be unique to that cube and reproducible at need. There is a similar requirement in the noise function, which also uses a cubic lattice with fixed values associated with each gridpoint. The solution to this problem is to hash the three integer coordinates of a cube into a single 32 bit integer which is used as the seed for a fast random number generator. One simple (but poor) hash function of three variables (i, j, k) is a linear combination like $541i + 79j + 31k \bmod 2^{32}$, but a much better (less correlated) method uses permutation arrays, such as the one described in ([1] pp 198.)

We find the number of points in the cube by using the first value from the seeded random number generator as an index into a list of the precomputed probabilities for different numbers of feature points. This is a binary search needing just couple of comparison tests to identify m.

Next, we compute the locations of the m feature points. Again, these are values that are random, but fixed for each cube. We use the already initialized random number generator to compute the XYZ locations of each of the feature points. These coordinates are relative to the base of the cube, and always range from 0..1 for each coordinate.

As we generate these points, we compute its distance to the original function evaluation location x, and keep a sorted list of the of the n smallest distances we've seen so far. As we test each point in turn, we effectively do an insertion sort to include the new point in the current list. This sounds expensive, but for typical values of n of 1 or 2, these cases are hard-wired tests of only one or two comparisons.

This finds the closest feature points and the values of F_n for the points within the current cube of space. However, the feature points within a neighboring cube could quite possibly contain a feature point even closer than the ones we have found already, so we must iterate among the boundary cubes too. Testing each of 26 boundary cubes would be slow, but by checking the closest distance we've computed so far (our tentative nth closest feature distance) we can throw out whole rows of cubes at once by noting that no point in the cube could possibly contribute to our list. Typically, only 1–3 cubes actually need to be tested.

Note that when we compute F_n we are effectively finding values for $F_1, F_2, ..., F_n$ simultaneously. In practice our routine returns all these values, plus the direction vectors corresponding to each feature point, plus a unique ID integer for each point (equal to the hashed cube ID plus the index of the feature point as it was computed). These tend to be useful when using the function to form solid textures.

In practice, computation is extremely efficient. A fast, linear congruential (LCG) random number generator is effectively just an integer multiply and add. By using fixed point arithmetic, the 32 bit random number can be manipulated directly. We avoid square roots by sorting on squared distance. Testing a point requires generation of its coordinates (three multiplies), computing the squared distance to the sample location (three multiplies), and insertion into the best candidate list (usually one to three compares.) The computation speed is therefore surprisingly fast. In our implementation, simultaneously computing F_1 and F_2 requires about the same amount of time as computing one scale of Perlin's noise.

We also found that by varying the point density λ we were able to tune our algorithm. Low λ requires fewer points to be computed in each cube. Higher λ makes it more likely to find the best points in the initial cube, reducing the number of neighboring cubes that must also be tested. We can choose λ to optimize speed, since the final point density can be manipulated at evaluation by simply scaling x before $F_n(\mathbf{x})$ is evaluated.Our implementation was most efficient at about $\lambda = 3$.

We lost some isotropy with the decision to forbid empty cubes. This was done to solve the problem of having such a sparse set of points in the cubes that huge numbers of neighbors may have to be evaluated to find all potential candidates. This is mainly a problem with low λ, since higher densities quickly reduce the average number of neighbors that need to be visited.

APPLICATION TO TEXTURING

The effort in implementing a function to compute $F_n(\mathbf{x})$ is rewarded by its extreme effectiveness as a solid texturing primitive. As with the Perlin noise function, mapping values of the function into a color and normal-displacement can produce visually interesting and impressive effects. A dense collection of ways to use the noise function can be found in [1]; since this new texturing basis function has the same functional form, it can be used in similar ways (but with different appearances.)

In the simplest case, $F_1(\mathbf{x})$ can be mapped into a color spline and bump. The character of F_1 is very simple, since the function increases radially around each feature point. Thus, mapping a color to small values of F_1 will cause a surface texture to place spots around each feature point; polka dots! Figure 1 shows this radial behavior in the upper left corner.

More interesting behavior begins when we start using the functions F_2 and F_3, shown as grey gradients in the upper right and lower left of Figure 1. These have more rapid changes and internal structure, and are slightly more visually interesting. These can be directly mapped into colors and bumps, but they can also produce even more interesting patterns by forming linear combinations with each other. For example, since $F_2 \geq F_1$ for all $\mathbf{x}$, we can map the function $F_2(\mathbf{x}) - F_1(\mathbf{x})$ to colors and bumps. The bottom right of Figure 1 shows this as a grey scale. This combination has a value of

0 where $F_1 = F_2$, which occurs at the Voronoi boundaries. This allows an interesting way to make a latticework of connected ridges, forming a vein-like tracery.

If the F_1 function returns a unique ID number to represent the closest feature point's identity, this number can be used to form values that are constant over a Voronoi cell; for example to shade the entire cell a single constant color. When combined with bumping based on $F_2 - F_1$, quite interesting surfaces can be easily generated. Figure 2 shows this technique, which also uses fractal noise discoloration in each cell. Note that unlike [4], no precomputation is necessary and the surface can be applied at any 3D object.

Bump mapping of the flagstone-like areas is particularly effective, and it is cheap to add since the gradient of F_n is just the radial unit vector pointing away from the appropriate feature point towards the sample location. Making raised spots or inset channels is done exactly like noise-based textures; ([1], pp105–110) has a useful discussion of applicable bump mapping methods.

We have slightly interesting patterns in the basis functions F_1, F_2, F_3 and now we see that the linear combination $F_2 - F_1$ forms a yet another basis. This leads us to experiment with other linear combinations, such as $2F_3 - F_2 - F_1$ or $F_1 + F_2$. In our first experiments, we generated about 40 different linear combinations to evaluate their appearance. We find that F_4 and other high n start looking similar, but the lower values of n (up to 4) are quite interesting and distinct. More importantly, linear combinations of these F_n have more "character" than the plain F_n, particularly differences of two or more simple bases. Figure 4 shows twenty sample surfaces which are all just examples of combinations of these low n basis functions ($C_1F_1 + C_2F_2 + C_3F_3 + C_4F_4$ for various values of C_n). We found that it was easy to try dozens of combinations simultaneously by mapping the value of each combination into a generic color spline. After empirically determining the range of values that the combination returns (by evaluating it at several thousand locations), we normalized this range to fall approximately between 0 and 1 for easier use as a texturing primitive. This process was only done once for each combination, after which it was simply hard-wired into the primitive basis.

These patterns are interesting and useful, but we can also use the basis functions to make *fractal* versions, much like noise is used to produce fractal noise. By computing multiple "scales" of the function at different weights and scaling factors, a more visually complex appearance can be made. This is a simple loop, computing a function $G_n = \sum 2^{-i} F_n(2^i \mathbf{x})$ for moderate ranges of i ($i = 0{-}5$), and using G_n as the index for colors and bumps.

The fractal versions of any of the basic basis function combinations become extremely appealing. Figure 5 shows a fractal version of F_1 forming the spotted pattern and bumps on the hide of a creature. Fractal noise is used for the tongue, and a linear gradient is applied to the main body for color variation. Other fractal versions of primitives are shown in the row of cut tori in Figure 6.

The fractal version of F_1 is perhaps the most useful. Applied solely as bump map, the surface becomes crumpled like paper or tinfoil. This surface has been extremely popular with artists as a way to break up a smooth surface, providing a subtle roughening with an appearance unlike fractal noise bumps. A surprising discovery was that a reflective, bumped map plane with this "crumple" appearance bears an excellent resemblance to seawater, as shown in Figure 7.

A variation of the algorithm uses different distance metrics. Using the Manhattan distance metric forms regions that are rigidly rectangular, but still random. These make surfaces like random right angle channels; useful for space ship hulls. Figure 3 shows a non-fractal version of F_1 which uses this Manhattan distance metric. A radial coordinate version can cover spheres, creating a surface similar to the "Death Star."

Other variations of the basic algorithms can produce even more effects. The density of the feature points can be made to vary spatially, allowing for small, dense features in one area and larger features in another. Object geometry might be used to disperse precomputed feature spots (similar to Turk[10]) at the expense of requiring precomputation and table lookup, but gaining object surface dependence similar to the advantages Turk found. The algorithm is normally computed in 3D, but 2D variants are even faster. 4D variants can be used for animated fields, though we find this to become significantly slower to compute by about a factor of 10.

CONCLUSION

We have found that this new texturing basis function is extremely useful in practical texture design. We have been using it in commercial products for several years and it is now an essential part of our texturing toolkit. It complements Perlin fractal noise; in any algorithm that uses noise, the new basis can be substituted. The visual effects are not necessarily similar, but this is desirable since it increases the visual diversity of the possible images.

ACKNOWLEDGEMENTS

Thanks to Richard Payne for his Gator and to Greg Teegarden for his water renderings. Particular thanks go to the referees with their excellent suggestions which made this a significantly better paper.

References

[1] EBERT, D. E. *Texturing and Modeling: A Procedural Approach*. Academic Press, 1994.

[2] GARDNER, G. Y. Simulation of natural scenes using textured quadric surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), H. Christiansen, Ed., vol. 18, pp. 11–20.

[3] LEWIS, J.-P. Algorithms for solid noise synthesis. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (July 1989), J. Lane, Ed., vol. 23, pp. 263–270.

[4] MIYATA, K. A method of generating stone wall patterns. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (Aug. 1990), F. Baskett, Ed., vol. 24, pp. 387–394.

[5] PEACHEY, D. Solid texturing of complex surfaces. In *Computer Graphics (SIGGRAPH '85 Proceedings)* (July 1985), B. A. Barsky, Ed., vol. 19, pp. 279–286.

[6] PEACHEY, D. Writing renderman shaders. In *1992 Course 21 Notes*. ACM SIGGRAPH, 1992.

[7] PERLIN, K. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)* (July 1985), B. A. Barsky, Ed., vol. 19, pp. 287–296.

[8] PERLIN, K., AND HOFFERT, E. M. Hypertexture. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (July 1989), J. Lane, Ed., vol. 23, pp. 253–262.

[9] SCHACHTER, B. J., AND AHUIA, N. Random pattern generation processes. *Computer Graphics and Image Processing 10* (1979), 95–114.

[10] TURK, G. Generating textures for arbitrary surfaces using reaction-diffusion. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (July 1991), T. W. Sederberg, Ed., vol. 25, pp. 289–298.