Cellular noise in GLSL
Implementation notes

Stefan Gustavson[1] 2011-04-19

[1]Media and Information Technology, Linköping University, Sweden
(stefan.gustavson@liu.se)

General notes

Cellular noise, often called Worley noise after Stephen Worley who presented his "cellular texture basis function" in 1996, is a useful tool for creating procedural textures. The patterns produced are of a different class than those that can be created with Perlin noise, and they have become popular with shader artists. Worleys original implementation is very much focused on execution on a regular CPU and does not map well to the massively parallel shader environment in a current GPU. These application notes describe my GLSL implementation of cellular noise, where several simplifications trade some generality and quality for significant increases in speed.

Simplifications

Exactly one point in each grid cell

Worleys original algorithm involves placing a pseudo-random number of pseudo-randomly located feature points in each cube of a regular 3D grid. For each input point $\vec{p}$, the algorithm determines the n closest feature points and returns their distances $\{F_1, F_2, \ldots, F_n\}$. The varying number of points in each cube is an obstacle to a parallel implementation. A simplification that has been popular in software shading because it can be efficiently expressed in RenderMan SL is to use exactly one feature point in each cube. This is similar to pseudo-random sample jittering, and maps better to a SIMD parallel execution because the same computations are performed at all points. Even though this simplified algorithm creates a different statistical distribution of points with some flaws, the generated patterns are visually very similar in nature.

2D instead of 3D

Another simplification is to define the feature points over a 2D grid of squares instead of a 3D grid of cubes, which significantly reduces the complexity of the search for the closest neighbors. The resulting 2D distance field is different from

a 2D slice of the 3D distance field, but it is still useful. With the addition of a z displacement for each feature point, the pattern can be made to look similar to a planar 2D slice of a 3D field, without the expense of a full 3D algorithm.

Smaller search region

The search for the closest feature points have traditionally been performed in a $3 \times 3 \times 3$ neighborhood of each cube. Such a large search region is important when n is large, but most real world uses of Worley noise compute only the closest and second closest neighbors, F_1 and F_2. Many patterns can in fact be designed by knowing only the closest distance, F_1. If F_2 is of less importance, the search for the closest feature points can be restricted to a 2×2 neighborhood in 3D, and a $2 \times 2 \times 2$ neighborhood in 2D. Both of these simplifications reduce the number of points that need to be searched, which speeds up the algorithm significantly. However, a 2×2 neighborhood in 2D makes F_2 almost useless as it is very often wrong and contains sharp discontinuities. A $2 \times 2 \times 2$ neighborhood in 3D will not be quite as bad for F_2, because there are more neighbors to search and a lower likelihood of the second closest neighbor being outside of the search region. By reducing the maximum displacement of each feature point within its corresponding cube (reducing the jitter), the errors in F_2 can be reduced to levels that could be tolerable, depending on the application. F_1 is mostly correct for a 2×2 search in 2D, and almost always correct for a $2 \times 2 \times 2$ search in 3D. In any case, the errors for F_1 are small and localized. If only F_1 is required, the smaller search regions are definitely worth considering.

Implementation details

Permutation polynomials

My implementation uses permutation polynomials for all pseudo-random numbers, using an idea by Ian McEwan and implemented like in the Perlin noise functions for GLSL published by him and me:
`http://github.com/ashima/webgl-noise`
On current generation GPUs, using a texture lookup into a pre-computed table for pseudo random index generation is generally faster, but less convenient.

Perturbations and distance computations

The implementation of perturbations and distance computations is very straightforward: First, associate a pseudo-random number with each integer spaced grid cell, and perturb one feature point from its original x position at the centre of the grid cell with a pseudo-random distance $-0.5 < d_x \leq 0.5$, and similarly for the other coordinates. Then, compute the distances from the input point to each perturbed feature point of the nearby grid cells.

Incomplete sorting

Worleys original implementation sorts the distances incrementally as they are found, and uses an early exit strategy to avoid unnecessary computations. This does not map well to a parallel execution. Instead, we first compute the distances to all feature points and then sort them. Determining just the smallest distance is a simple task that merely involves repeated use of the min function in GLSL, which parallelizes nicely by using vector arguments. If only F_1 is of interest, this simpler sorting is quick and efficient.

Sorting out both the smallest and the second smallest value is a bit more difficult to do in GLSL. Standard sorting algorithms use a compare and swap strategy, and while comparisons can be made easily and in parallel by the `lessThan` function and its siblings, there is no swap function for general arguments in GLSL. Combined use of the `?` operator and swizzles can be used to swap components within a vector. The `lessThan` and `mix` functions could be combined for a per-component swap between vectors, but it seems a waste of resources to perform the inherent multiplications of the `mix` function with constant factors 0 and 1 just to pick one value or the other. Instead, I have used combinations of `min` and `max` functions to perform swaps. While that ends up making the same comparison twice, it seems to be generally faster than one comparison and one `mix` on current GPU platforms.

The sorting is deliberately incomplete. A full sort would be terribly wasteful, particularly for the larger search regions, becasuse only the two smallest values are of interest. By careful consideration, there are points in the sorting where certain values can be determined to be neither the smallest nor the second smallest of the set. From that point on, these values can be discarded from further comparisons. By similar reasoning, swapping can be replaced with simple copying when the overwritten values do not need further consideration.

Source code and performance

Source code for the functions, along with the most recent version of this document, is provided on the following address:
`http://www.itn.liu.se/~stegu/GLSL-cellular/`
The source code is distributed under the terms of the very permissive MIT license. Performance of these functions are good. The 3D version with the large search region of cells involves a lot of computations and is considerably slower than the others, but the rest of the functions can be considered for routine use even on older and less powerful GPUs. Benchmarking on a current mid-range Nvidia GTX260 GPU gave the following results:

cellular2D	855 M samples per second
cellular2x2	1700 M samples per second
cellular3D	315 M samples per second
cellular2x2x2	920 M samples per second

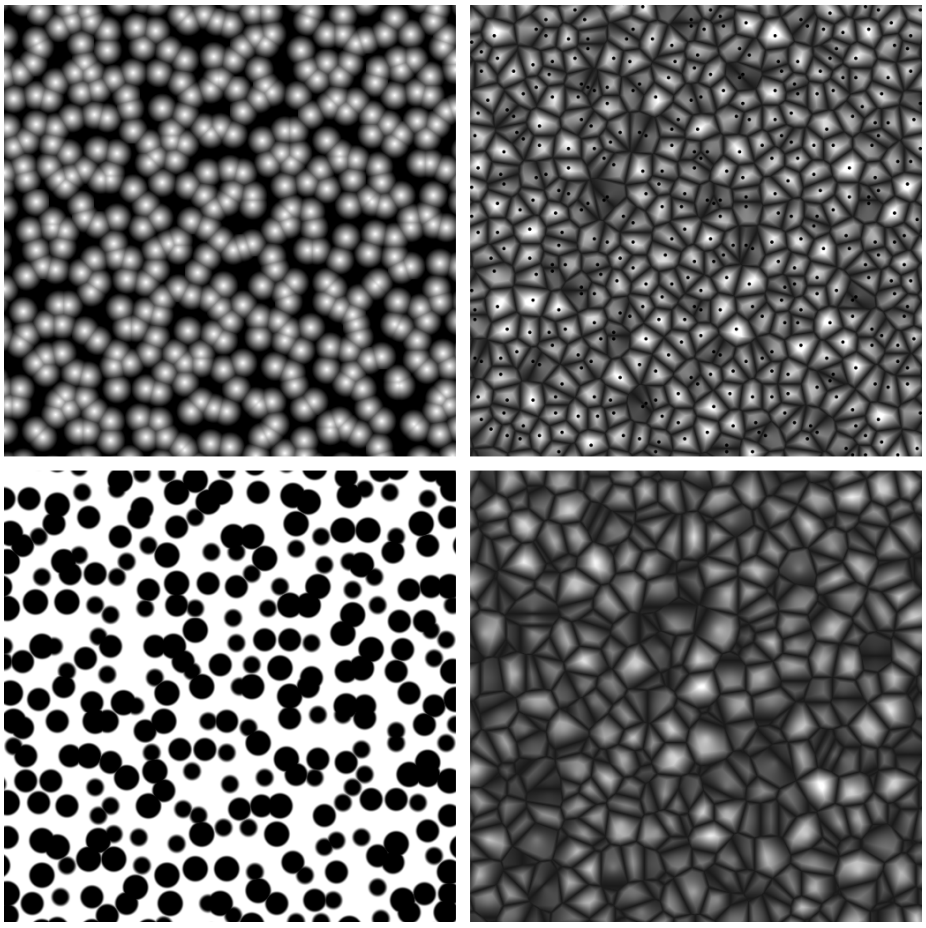

Figure 1: Cellular noise patterns. **Top left:** 2-D 2×2 version using F_1 only. **Top right:** 2-D 3×3 version using F_1 and F_2. **Bottom left:** 3-D $2 \times 2 \times 2$ version using F_1 only. **Bottom right:** 3-D $3 \times 3 \times 3$ version using F_1 and F_2.