

## CMSC 425: Lecture 13

### Procedural Generation: 2D Perlin Noise

**Reading:** The material on Perlin Noise based in part by the notes *Perlin Noise*, by Hugo Elias. (The link to his materials seems to have been lost.)

**Perlin Noise in 2D:** In the previous lecture we introduced the concept of Perlin noise, a structured random function, in a one-dimensional setting. In this lecture we show how to generalize this concept to a two-dimensional setting. Such two-dimensional noise functions can be used for generating pseudo-random terrains and two-dimensional pseudo-random textures.

The general approach is the same as in the one-dimensional case:

- Generate a finite sample of random values
- Generate a noise function that interpolates smoothly between these values
- Sum together various octaves of this function by scaling it down by factors of  $1/2$ , and then applying a dampening persistence value to each successive octave, so that high frequency variations are diminished

Let's investigate the finer points. First, let us begin by assuming that we have an  $n \times n$  grid of unit squares (see Fig. 1(a)), for a relatively small number  $n$  (e.g.,  $n$  might range from 2 to 10). For each vertex  $[i, j]$  of this grid, where  $0 \leq i, j \leq n$ , let us generate a random scalar value  $z_{[i,j]}$ . (Note that these values are actually not very important. In Perlin's implementation of the noise function, these values are all set to 0, and it still produces a remarkably rich looking noise function.) As in the 1-dimensional case, it is convenient to have the values wrap around, which we can achieve by setting  $z_{[i,n]} = z_{[i,0]}$  and  $z_{[n,j]} = z_{[0,j]}$  for all  $i$  and  $j$ .

Given any point  $(x, y)$ , where  $0 \leq x, y < n$ , the corner points of the square containing this point are  $(x_0, y_0)$ ,  $(x_1, y_0)$ ,  $(x_1, y_1)$ ,  $(x_0, y_1)$  where:

$$\begin{aligned} x_0 &= \lfloor x \rfloor & \text{and} & & x_1 &= (x_0 + 1) \bmod n \\ y_0 &= \lfloor y \rfloor & \text{and} & & y_1 &= (y_0 + 1) \bmod n \end{aligned}$$

(see Fig. 1(a)).

We could simply apply smoothing to the random values at the grid points, but this would produce a result that clearly had a rectangular blocky look (since every square would suffer the same variation). Instead, Perlin came up with a way to have every vertex behave differently, by creating a random gradient at each vertex of the grid.

**Noise from Random Gradients:** Before explaining the concept of gradients, let's recall some basics from differential calculus. Given a continuous function  $f(x)$  of a single variable  $x$ , we know that the derivative of the function  $df/dx$  yields the tangent slope at the point  $(x, f(x))$  on the function. If we instead consider a function  $f(x, y)$  of two variables, we can visualize the function values  $(x, y, f(x, y))$  as defining the height of a point on a two-dimensional terrain. If  $f$  is smooth, then each point of the terrain can be associated with tangent plane. The "slope" of the tangent plane passing through such a point is defined by the partial derivatives of the

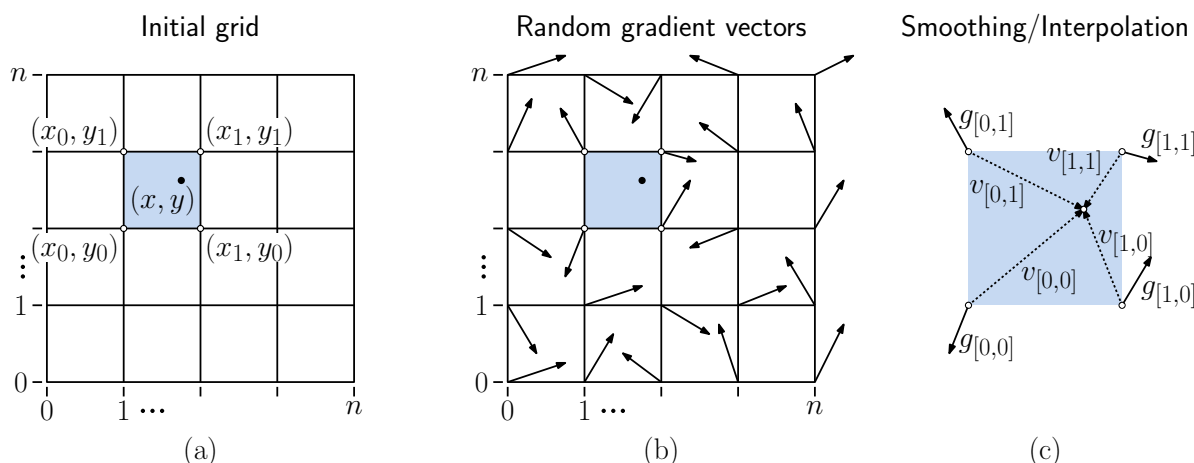


Fig. 1: Generating 2-dimensional Perlin noise.

function, namely  $\partial f/\partial x$  and  $\partial f/\partial y$ . The vector  $(\partial f/\partial x, \partial f/\partial y)$  is a vector in the  $(x, y)$ -plane that points in the direction of *steepest ascent* for the function  $f$ . This vector changes from point to point, depending on  $f$ . It is called the *gradient* of  $f$ , and is often denoted  $\nabla f$ .

Perlin's approach to producing a noisy 2-dimensional terrain involves computing a random 2-dimensional gradient vector at each vertex of the grid with the eventual aim that the smoothed noise function have this gradient value. Since these vectors are random, the resulting noisy terrain will appear to behave very differently from one vertex of the grid to the next. At one vertex the terrain may be sloping up to the northeast, and at a neighboring vertex it may be sloping to south-southwest. The random variations in slope result in a very complex terrain. But how do we define a smooth function that has this behavior? In the one dimensional case we used cosine interpolation. Let's consider how to generalize this to a two-dimensional setting.

Consider a single square of the grid, with corners  $(x_0, y_0)$ ,  $(x_1, y_0)$ ,  $(x_1, y_1)$ ,  $(x_0, y_1)$ . Let  $g_{[0,0]}$ ,  $g_{[1,0]}$ ,  $g_{[1,1]}$ , and  $g_{[0,1]}$  denote the corresponding randomly generated 2-dimensional gradient vectors (see Fig. 1(c)). Now, for each point  $(x, y)$  in the interior of this grid square, we need to blend the effects of the gradients at the corners. To do this, for each corner we will compute a vector from the corner to the point  $(x, y)$ . In particular, define

$$\begin{aligned} v_{[0,0]} &= (x, y) - (x_0, y_0) & \text{and} & & v_{[0,1]} &= (x, y) - (x_0, y_1) \\ v_{[1,0]} &= (x, y) - (x_1, y_0) & \text{and} & & v_{[1,1]} &= (x, y) - (x_1, y_1) \end{aligned}$$

(see Fig. 1(c)).

Next, for each corner point of the square, we generate an associated *vertical displacement*, which indicates the height of the point  $(x, y)$  due to the effect of the gradient at this corner point. How should this displacement be defined? Let's fix a corner, say  $(x_0, y_0)$ . Intuitively, if  $v_{[0,0]}$  is directed in the same direction as the gradient vector, then the vertical displacement will increase (since we are going uphill). If it is in the opposite direction, the displacement will decrease (since we are going downhill). If the two vectors are orthogonal, then the vector  $v_{[0,0]}$  is directed neither up- or downhill, and so the displacement is zero. Among the vector

operations we have studied, the *dot product* produces exactly this sort of behavior. (When two vectors are aligned, the dot-product is maximized, when they are anti-aligned it is minimized, and it is zero when they are orthogonal. It also scales linearly with the length, so that a point that is twice as far away along a given direction has twice the displacement.) With this in mind, let us define the following scalar displacement values:

$$\begin{aligned}\delta_{[0,0]} &= (v_{[0,0]} \cdot g_{[0,0]}) & \text{and} & & \delta_{[0,1]} &= (v_{[0,1]} \cdot g_{[0,1]}) \\ \delta_{[1,0]} &= (v_{[1,0]} \cdot g_{[1,0]}) & \text{and} & & \delta_{[1,1]} &= (v_{[1,1]} \cdot g_{[1,1]}).\end{aligned}$$

**Fading:** The problem with these scalar displacement values is that they are affected by all the corners of the square, and in fact, as we get farther from the associated corner point the displacement gets larger. We want the gradient effect to apply close to the vertex, and then have it drop off quickly as we get closer to another vertex. That is, we want the gradient effect of this vertex to *fade* as we get farther from the vertex. To do this, Perlin defines the following *fade function*. This is a function of  $t$  that will start at 0 when  $t = 0$  (no fading) and will approach 1 when  $t = 1$  (full fading). Perlin originally settled on a cubic function to do this,  $\varphi(t) = 3t^2 - 2t^3$ . (Notice that this has the desired properties, and further its derivative is zero at  $t = 0$  and  $t = 1$ , so it will smoothly interpolate with neighboring squares.) Later, Perlin observed that this function has nonzero second derivatives at 0 and 1, and so he settled on the following improved fade function:

$$\psi(t) = 6t^5 - 15t^4 + 10t^3$$

(see Fig. 2). Observe again that  $\psi(0) = 0$  and  $\psi(1) = 1$ , and the first and second derivatives are both zero at these endpoints.

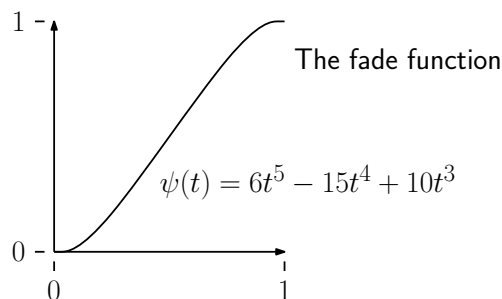


Fig. 2: The fade function.

Because we want the effects to fade as a function of both  $x$  and  $y$ , we define the *joint fade function* to be the product of the fade functions along  $x$  and  $y$ :

$$\Psi(x, y) = \psi(x)\psi(y).$$

The final noise value at the point  $(x, y)$ , arises by taking the weighted average of gradient displacements, where each displacement is weighted according to the fade function.

We need to apply the joint fade function differently for each vertex. For example, consider the fading for the displacement  $\delta_{[1,0]}$  of the lower right corner vertex. We want the influence

of this vertex to increase as  $x$  approaches 1, which will be achieved by using a weight of  $\psi(x)$ . Similarly, we want the influence of this vertex to increase as  $y$  approaches 0, which will be achieved by using a weight of  $\psi(1 - y)$ . Therefore, to achieve both of these effects, we will use the joint weight function  $\Psi(x, 1 - y)$ . By applying this reasoning to the other corner vertices, we obtain the following 2-dimensional noise function.

$$\text{noise}(x, y) = \Psi(1 - x, 1 - y)\delta_{[0,0]} + \Psi(x, 1 - y)\delta_{[1,0]} + \Psi(1 - x, y)\delta_{[0,1]} + \Psi(x, y)\delta_{[1,1]}.$$

**Adding Back the Random Heights:** We have left one little bit out of our noise function. Remember that we started off by assigning random scalar values to each of the of the grid. We never made use of these (and indeed, Perlin’s formulation of the noise function does not either). In order to achieve this extra degree of randomness, we can add these back into the vertical displacements. Suppose, for example that we are considering the grid square whose lower left corner has the indices  $[i, j]$ . When defining the vertical displacements, let us add in the associated random scalar values associated with each of the vertices:

$$\begin{aligned} \delta_{[0,0]} &= z_{[i,j]} + (v_{[0,0]} \cdot g_{[0,0]}) & \text{and} & & \delta_{[0,1]} &= z_{[i,j+1]} + (v_{[0,1]} \cdot g_{[0,1]}) \\ \delta_{[1,0]} &= z_{[i+1,j]} + (v_{[1,0]} \cdot g_{[1,0]}) & \text{and} & & \delta_{[1,1]} &= z_{[i+1,j+1]} + (v_{[1,1]} \cdot g_{[1,1]}). \end{aligned}$$

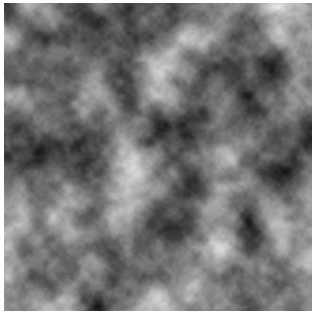
The rest of the noise computation is exactly the same as described above.

**Octaves and Persistence:** After all of that work, we still have only a single smooth noise function, but not one that demonstrates the sort of fractal-like properties we desire. To add this, we need to perform the same scaling that we used for the 1-dimensional case. In this case, the process is entirely analogous. As before, let  $p$  be a value between 0 and 1, which will determine how quickly things dampen down. Also, as before, at each level of the process, we will double the frequency. This leads to the following final definition of the 2-dimensional Perlin noise:

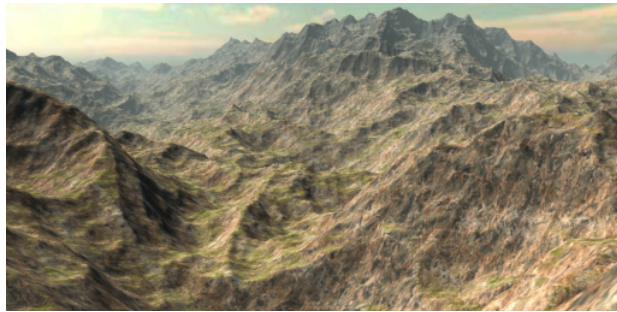
$$\text{perlin}(x, y) = \sum_{i=0}^k p^i \cdot \text{noise}(2^i \cdot x, 2^i \cdot y).$$

(As before, recall that the value  $2^i$  can be replaced by some parameter  $\ell^i$ , where  $\ell > 1$ .) This applies to each square individually. We need to perform the usual “modding” to generalize this to any square of the grid. (An example of the final result is shown in Fig. 3(a) and a terrain resulting from applying this is shown in Fig. 3(b). Note that the terrain does not look as realistic as the *terrain* from Fig. 1(a). There are other processes, such as erosion and geological forces that need to be modeled to achieve highly realistic terrains.)

**Source Code:** While the mathematical concepts that we have discussed are quite involved, it is remarkable that Perlin noise has a very simple implementation. The entire implementation can be obtained from Perlin’s web page, and is shown nearly in its entirety in the code block below.



(a)



(b)

Fig. 3: (a) Two-dimensional Perlin noise and (b) a terrain generated by Perlin noise.

## Perlin's Implementation of Perlin Noise

---

```

public final class ImprovedNoise {
    static public double noise(double x, double y, double z) {
        int X = (int)Math.floor(x) & 255, // Fine unit cube that
            Y = (int)Math.floor(y) & 255, // contains point
            Z = (int)Math.floor(z) & 255;

        x -= Math.floor(x); // Find relative x,y,z
        y -= Math.floor(y); // of point in cube
        z -= Math.floor(z);

        double u = fade(x), // Compute fade curves
               v = fade(y), // for each of x,y,z
               w = fade(z);

        int A = p[X ]+Y, AA = p[A]+Z, AB = p[A+1]+Z, // Hash coordinates of
            B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z; // the 8 cube corners
        // ...and add blended results from 8 corners of cube
        return lerp(w, lerp(v, lerp(u, grad(p[AA ], x , y , z ),
                                           grad(p[BA ], x-1, y , z )),
                    lerp(u, grad(p[AB ], x , y-1, z ),
                           grad(p[BB ], x-1, y-1, z ))),
                lerp(v, lerp(u, grad(p[AA+1], x , y , z-1 ),
                                   grad(p[BA+1], x-1, y , z-1 )),
                    lerp(u, grad(p[AB+1], x , y-1, z-1 ),
                           grad(p[BB+1], x-1, y-1, z-1 ))));
    }

    static double fade(double t) { return t*t*t*(t*(t*6 - 15) + 10); }
    static double lerp(double t, double a, double b)
        { return a + t*(b - a); }
    static double grad(int hash, double x, double y, double z) {
        int h = hash & 15; // Convert low 4 bits of hash code
        double u = h<8 ? x : y, // into 12 gradient directions
               v = h<4 ? y : h==12||h==14 ? x : z;
        return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
    }

    static final int p[] = new int[512], permutation[] = {
        151,160,137,91,90,15, // ... remaining 506 entries omitted
    };
    static { for (int i=0; i < 256 ; i++)
        p[256+i] = p[i] = permutation[i]; }
}

```

---