CMSC 425: Lecture 12
Procedural Generation: 1D 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.) This is not exactly the same as Perlin noise, but the principles are the same.

Procedural Generation: Complex AAA games hire armies of designers to create the immense content that make up the game's virtual world. If you are designing a game without such extensive resources, an attractive alternative for certain natural phenomena (such as terrains, trees, and atmospheric effects) is through the use of *procedural generation*. With the aid of a random number generator, a high quality procedural generation system can produce remarkably realistic models. Examples of such systems include *terragen* (see Fig. 1(a)) and *speedtree* (see Fig. 1(b)).

Fig. 1: (a) A terrain generated by *terragen* and (b) a scene with trees generated by *speedtree*.

Procedural model generation is a useful tool in developing open-world games. For example, the game *No Man's Sky* uses procedural generation to generate a universe with a vast number of different planets, all with distinct pseudo-randomly-generated ecosystems, including terrains, flora, fauna, and climates (see Fig. 2). The structure of each planet is not stored on a server. Instead, each is generated deterministically by a 64-bit seed.

Fig. 2: No Man's Sky.

Before discussing methods for generating such interesting structures, we need to begin with a background, which is interesting in its own right. The question is how to construct random noise that has nice structural properties. In the 1980's, Ken Perlin came up with a powerful and general method for doing this (for which he won an Academy Award!). The technique is now widely referred to as Perlin Noise.

Perlin Noise: Natural phenomena derive their richness from random variations. In computer science, pseudo-random number generators are used to produce number sequences that appear to be random. These sequences are designed to behave in a totally random manner, so that it is virtually impossible to predict the next value based on the sequence of preceding values. Nature, however, does not work this way. While there are variations, for example, in the elevations of a mountain or the curves in a river, there is also a great deal of structure present as well.

One of the key elements to the variations we see in natural phenomena is that the magnitude of random variations depends on the scale (or size) at which we perceive these phenomena. Consider, for example, the textures shown in Fig. 3. By varying the frequency of the noise we can obtain significantly different textures.

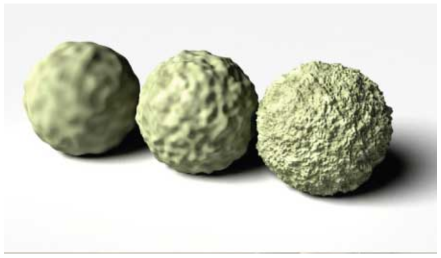

Fig. 3: Perlin noise used to generate a variety of displacement textures.

The tendency to see repeating patterns arising at different scales is called *self similarity* and it is fundamental to many phenomena in science and nature. Such structures are studied in mathematics under the name of *fractals*. Perlin noise can be viewed as a type of random noise that is self similar at different scales, and hence it is one way of modeling random fractal objects.

Noise Functions: Let us begin by considering how to take the output of a pseudo-random number generator and convert it into a smooth (but random looking) function. To start, let us consider a sequence of random numbers in the interval $[0, 1]$ produced by a random number generator (see Fig. 4(a)). Let $Y = \langle y_0, \ldots, y_n \rangle$ denote the sequence of random values, and let us plot them at the uniformly places points $X = \langle 0, \ldots, n \rangle$.

Next, let us map these points to a continuous function, we could apply linear interpolation between pairs of points (also called *piecewise linear interpolation.* As we have seen earlier this semester, in order to interpolate linearly between two values y_i and y_{i+1}, we define a parameter α that varies between 0 and 1, the interpolated value is

$$\mathrm{lerp}(y_i, y_{i+1}, \alpha) \;=\; (1 - \alpha)y_i + \alpha y_{i+1}.$$

To make this work in a piecewise setting we need to set α to the fractional part of the x-value that lies between i and $i+1$. In particular, if we define $x \bmod 1 = x - \lfloor x \rfloor$ to be the fractional

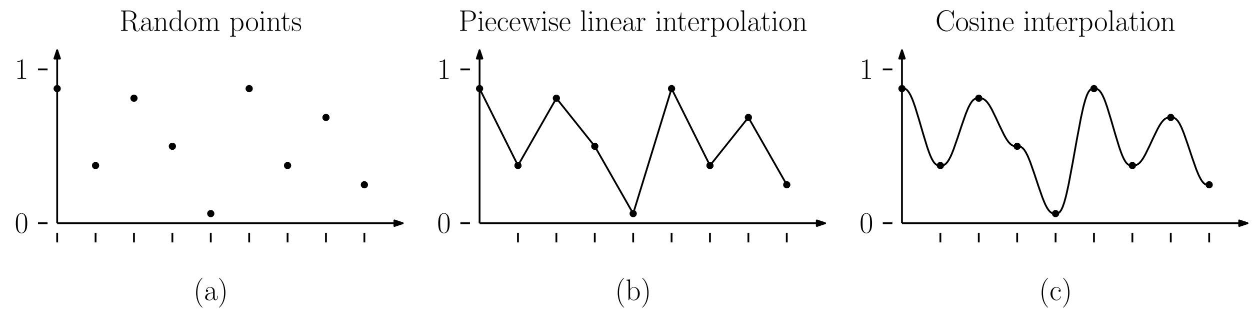

Fig. 4: (a) Random points, (b) connected by linear interpolation, and (c) connected by cosine interpolation.

part of x, we can define the linear interpolation function to be

$$f_\ell(x) \;=\; \mathrm{lerp}(y_i, y_{i+1}, \alpha), \qquad \text{where } i = \lfloor x \rfloor \text{ and } \alpha = x \bmod 1.$$

The result is the function shown in Fig. 4(b).

While linear interpolation is easy to define, it will not be sufficient smooth for our purposes. There are a number of ways in which to define smoother interpolating functions. (This is a topic that is usually covered in computer graphics courses.) A quick-and-dirty way to define such an interpolation is to replace the linear blending functions $(1-\alpha)$ and α in linear interpolation with smoother functions that have similar properties. In particular, observe that α varies from 0 to 1, the function $1-\alpha$ varies from 1 down to 0 while α goes the other way, and for any value of α these two functions sum to 1 (see Fig. 5(a)). Observe that the functions $(\cos(\pi\alpha)+1)/2$ and $(1-\cos(\pi\alpha))/2$ behave in exactly this same way (see Fig. 5(b)). Thus, we can use them as a basis for an interpolation method.

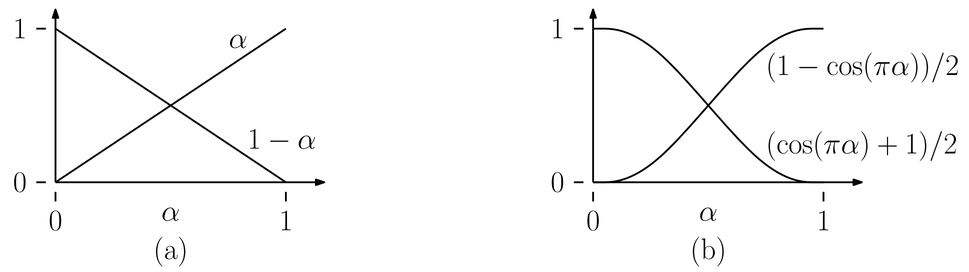

Fig. 5: The blending functions used for (a) linear interpolation and (b) cosine interpolation.

Define $g(\alpha) = (1-\cos(\pi\alpha))/2$. The cosine interpolation between two points y_i and y_{i+1} is defined:

$$\mathrm{cerp}(y_i, y_{i+1}, \alpha) \;=\; (1-g(\alpha))y_i + g(\alpha)y_{i+1},$$

and we can extend this to a sequence of points as

$$f_c(x) \;=\; \mathrm{cerp}(y_i, y_{i+1}, \alpha), \qquad \text{where } i = \lfloor x \rfloor \text{ and } \alpha = x \bmod 1.$$

The result is shown in Fig. 4(c). While cosine interpolation does not generally produce very good looking results when interpolating general data sets. (Notice for example the rather artificial looking flat spot as we pass through the fourth point of the sequence.) Interpolation methods such as cubic interpolation and Hermite interpolate are preferred. It is worth remembering, however, that we are interpolating random noise, so the lack of "goodness" here is not much of an issue.

Layering Noise: Our noise function is continuous, but there is no self-similarity to its structure. To achieve this, we will need to combine the noise function in various ways. Our approach will be similar to the approach used in the harmonic analysis of functions.

Recall that when we have a periodic function, like $\sin t$ or $\cos t$, we define (see Fig. 6)

Wavelength: The distance between successive wave crests

Frequency: The number of crests per unit distance, that is, the reciprocal of the wavelength

Amplitude: The height of the crests

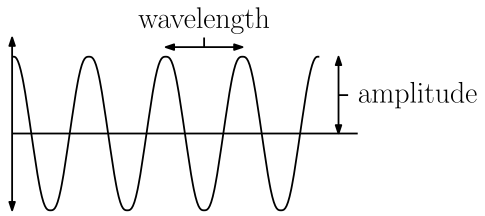

Fig. 6: Properties of periodic functions.

If we want to decrease the wavelength (equivalently increase the frequency) we can scale up the argument. For example $\sin t$ has a wavelength of 2π, $\sin(2t)$ has a wavelength of π, and $\sin(4t)$ has a wavelength of $\pi/2$. (By increasing the value of the argument we are increasing the function's frequency, which decreases the wavelength.) To decrease the function's amplitude, we apply a scale factor that is smaller than 1 to the value of the function. Thus, for any positive reals ω and α, the function $\alpha \cdot \sin(\omega t)$ has a wavelength of $2\pi/\omega$ and an amplitude of α.

Now, let's consider doing this to our noise function. Let $f(x)$ be the noise function as defined in the previous section. Let us assume that $0 \leq x \leq n$ and that the function repeats so that $f(0) = f(n)$ and let us assume further that the derivatives match at $x = 0$ and $x = n$. We can convert f into a periodic function for all $t \in \mathbb{R}$, which we call $\text{noise}(t)$, by defining

$$\text{noise}(t) \;=\; f(t \bmod n).$$

(Again we are using the mod function in the context of real numbers. Formally, we define $x \bmod n = x - n \cdot \lfloor x/n \rfloor$.) For example, the top graph of Fig. 7 shows three wavelengths of $\text{noise}(t)$.

In order to achieve self-similarity, we will sum together this noise function, but using different frequencies and with different amplitudes. First, we will consider the noise function with exponentially increasing frequencies: $\text{noise}(t)$, $\text{noise}(2t)$, $\text{noise}(4t)$, $\ldots$, $\text{noise}(2^i t)$ (see Fig. 8). Note that we have not changed the underlying function, we have merely modified its frequency. In the jargon of Perlin noise, these are called *octaves*, because like musical octaves, the frequency doubles.[1] Because frequencies double with each octave, you do not need very many octaves, because there is nothing to be gained by considering wavelengths that are larger than the entire screen nor smaller than a single pixel. Thus, the logarithm of the window size is a natural upper bound on the number of octaves.

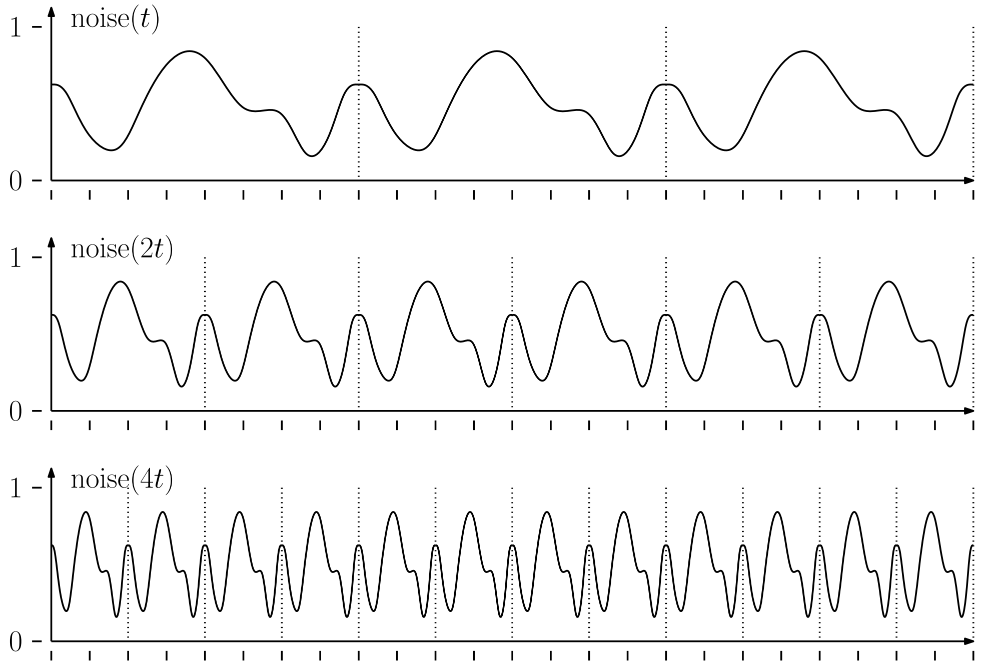

Fig. 7: The periodic noise function at various frequencies.

High frequency noise tends to be of lower amplitude. If we were in a purely self-similar situation, when the double the frequency, we should halve the amplitude. In order to provide the designer with more control, Perlin noise allows the designer to specify a separate amplitude for each frequency. A common way in which to do this is to define a parameter, called *persistence*, that specifies how rapidly the amplitudes decrease. Persistence is a number between 0 and 1. The larger the persistence value, the more noticeable are the higher frequency components. (That is, the more "jagged" the noise appears.) In particular, given a persistence of p, we define the amplitude at the ith stage to be p^i. The final noise value is the sum, over all the octaves, of the persistence-scaled noise functions. In summary, we have

$$\text{perlin}(t) \;=\; \sum_{i=0}^{k} p^i \cdot \text{noise}(2^i \cdot t),$$

where k is the highest-frequency octave.

It is possible to achieve greater control over the process by allowing the user to modify the octave scaling values (currently 2^i) and the persistence values (currently p^i).

[1]In general, it is possible to use factors other than 2. Such a factor is called the *lacunarity* of the Perlin noise function. For example, a lacunarity value of ℓ means that the frequency at stage i will be ℓ^i.

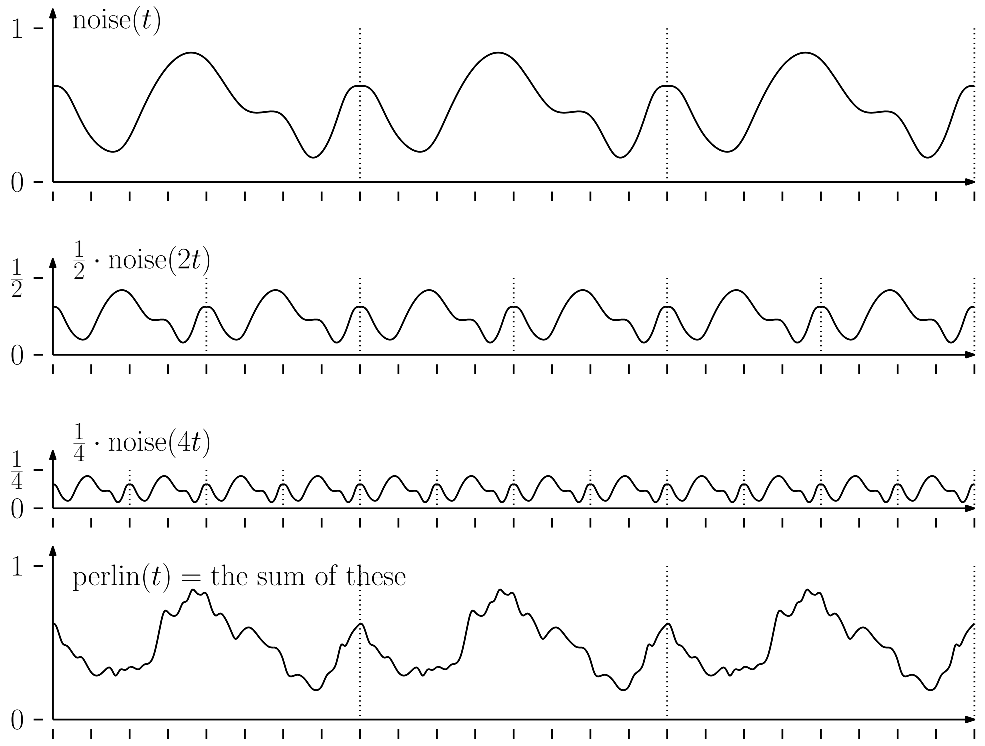

Fig. 8: Dampened noise functions and the Perlin noise function (with persistence $p = 1/2$).