Coherent Noise

User manual

Table of Contents

Introduction

Procedural generation has a long history with games. Game content might be too big to fit in memory. It might be too time-consuming to create, or too repetitive. For any of these reasons, or just for no reason at all, game developers used generated content. After all, why spend time on something that the computer can do instead?

Rogue had endless procedurally-generated dungeons. Civilization had a different map for each play-through. Elite had a whole galaxy of stars and planets, created on-the-fly.

One of the most powerful tools for procedural generation is coherent noise. Here's a small list of things that can be made with it:

- Clouds, smoke and fog
- All kinds of natural textures, including wood, tree bark, stone, marble, sand, dirt etc.
- Cave networks and dungeons
- Fields of grass
- Rolling hills and mountains
- Waves on water
- Realistic gestures and face expressions
- Road networks
- Islands and continents
- Starfields
- Space nebulae
- And entire universes full of different stars and planets

CoherentNoise is a library for Unity3D that allows the use of noise in your projects. It has methods to generate all kinds of noise, modify it and combine different noises together; all with a simple and clean object-oriented API. There

are also some helpful methods to create textures and materials directly from noise functions.

Chapter I of this manual explains the basics of noise generation and the basic principles of CoherentNoise API. Chapter II lists all the library's classes, detailing their usage. Chapter III presents a couple of examples, using noise to procedurally create textures.

Chapter I.

Noise Basics.

Everyone has an intuitive understanding of what noise is. Noise is when something sounds like "khrrshshhh..." This doesn't seem very useful for all the nice things mentioned in the introduction, though.

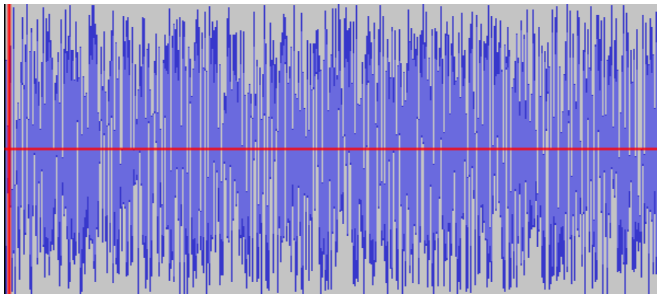


Fig. 1: Noise waveform

But let's have a closer look at this "khrrshshhh..." Figure 1 shows a waveform of audible noise. Note how the graph "jumps" up and down chaotically. The values on the graph seem to have no relation to each other. Let's define noise like this:

> *Noise is a function* $f(x)$*, such as any two values,* $f(x_0)$ *and* $f(x_1)$ *have no correlation unless* $x_0 = x_1$*.*

Here, we have a strict mathematical definition of noise! It still does not seem terribly useful.

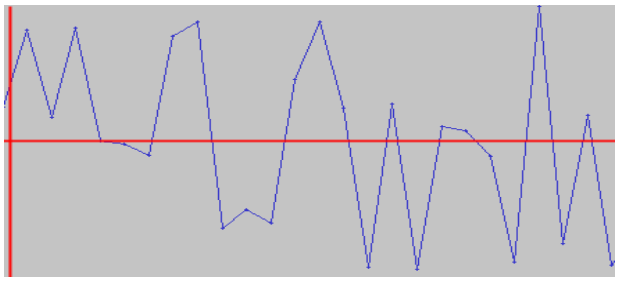


Fig. 2: Zoomed-in noise

This definition misses an important property of "natural" noise. Let's look even closer at the waveform. Figure 2 shows a zoomed-in variant... see how a close-up doesn't seem that chaotic? Natural noise is chaotic on large scale, but smooth on small scale. Armed with this knowledge, we can make another definition.

> *Coherent noise is a function* $f(x)$*, such as any two values* $f(x_0)$ *and* $f(x_1)$ *are close together when* x_0 *and* x_1 *are close together, but do not correlate when* x_0 *and* x_1 *are far apart.*

This definition is not very strict, but it will do for now.

With this definition we can actually construct a program to output noise. One widely-used approach for noise generation is so-called *lattice noise*; this is the approach taken by CoherentNoise library. Lattice noise works by calculating random numbers at lattice points - i.e. points with integer coordinates. An integer can be turned into a pseudo-random number by "mangling" its bits - the result will not be a true random number, but it will suffice. With a random number, we can calculate noise function value, $f(x)$, when x is integer. In CoherentNoise library this is done by simply scaling the value, so that it falls between -1 and 1.

These values ensure the second part of our definition: $f(x_0)$ and $f(x_1)$ are absolutely unrelated when $|x_0 - x_1| \geqslant 1$. To make the noise coherent - that is, satisfy the first part of the definition - we can interpolate noise values between lattice points. The resulting variant of lattice noise is called *value noise*.

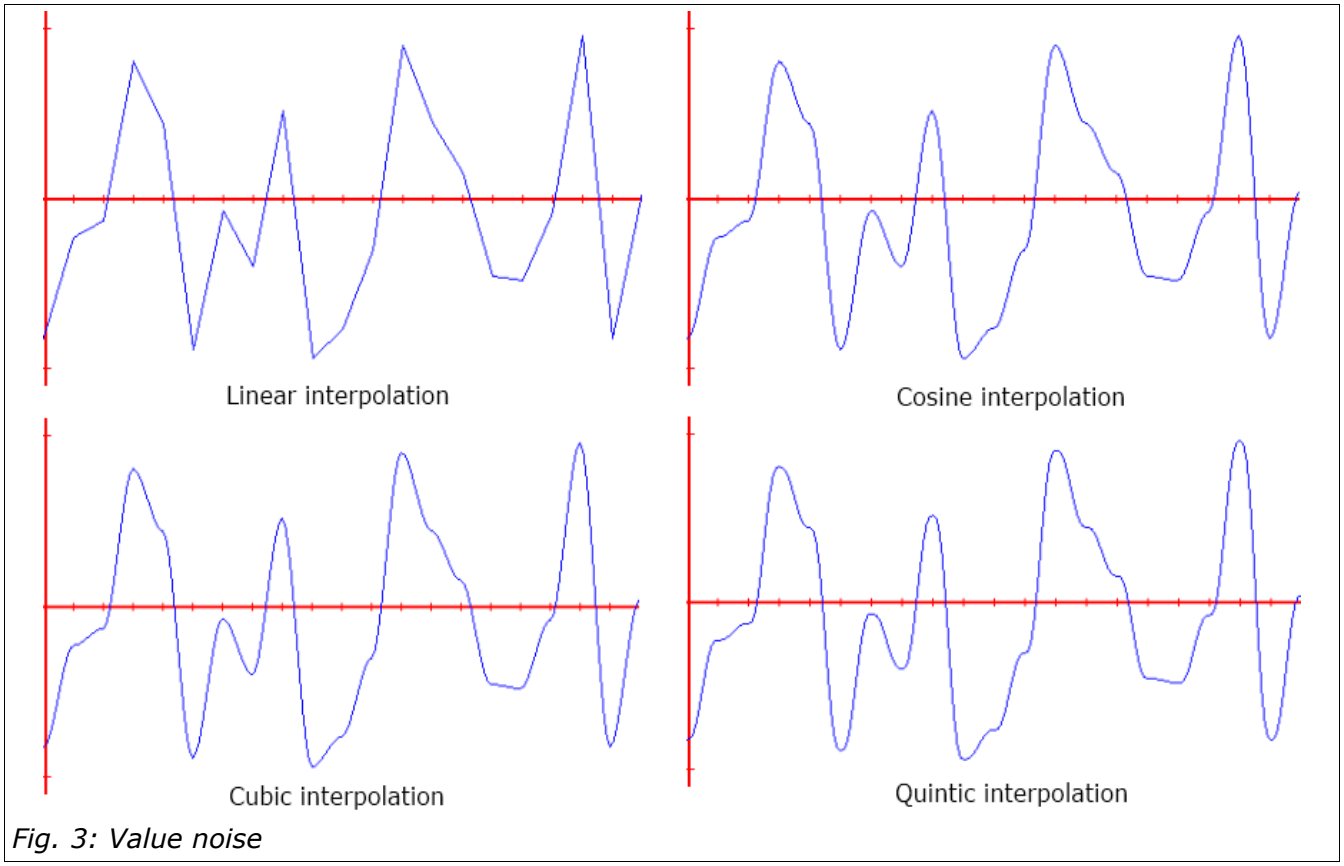


Fig. 3: Value noise

Value noise depends greatly on the interpolation method used. Linear interpolation is simple and fast, but it produces low-quality functions, that look like an irregular saw. To make noise better, there are other interpolation methods. Basically, any "S-shaped" curve can serve as an interpolator. CoherentNoise uses cubic, quintic and cosine curves for interpolation (see figure 3 for comparison), with cubic as a default.

While value noise function satisfies our definition, it still has a drawback. Looking closely at the graph, we can see that local minimum and maximum values are always at integer points. This means that noise will show noticeable "grid" artifacts. As a way to reduce these artifacts we can use a more clever function, called *gradient noise* (another name for this function is Perlin noise, after Ken Perlin who invented it in 1985).

Gradient noise is a variation of lattice noise. Instead of determining noise function value at integer points, in gradient noise random generator is used to find function gradient. The value of noise at integer points is set to 0, and then interpolated so that gradients match up. Resulting noise is shown in figure 4. This function still has a "regular" feature, as it is equal to 0 at integer points. However,

in practice this leads to less visual artifacts than value noise.

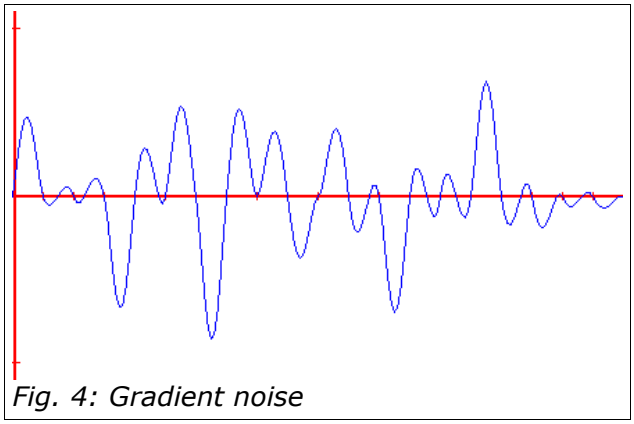


Fig. 4: Gradient noise

Other variants of noise exist. One can use different interpolation techniques, or a convolution filter over integer lattice, to achieve better, less regular functions. These functions tend to be quite slow, which is why CoherentNoise uses gradient noise as a default. One interesting variant that can be easily achieved with CoherentNoise is simply an average of value and gradient noise. This will break up both extrema pattern of value noise and zeroes pattern of gradient noise. An example of such noise is shown in figure 5.

Our definition, and the examples above, apply to one-dimensional noise - a function of one coordinate. But they can be extended trivially for more dimensions. By interpolating 4 values in 2D , or 8 values in 3D, we can get 2D and 3D noise functions. 2D noise is usually used for texture generation and heightmaps. 3D noise can represent a "solid texture", from which an object is then "carved", or an animated 2D texture with third dimension used as time, or a volumetric density map for fog or clouds. 4D noise is sometimes used as animated version of 3D noise. In CoherentNoise, all functions are 3D by default, with some optimized for 2D use.

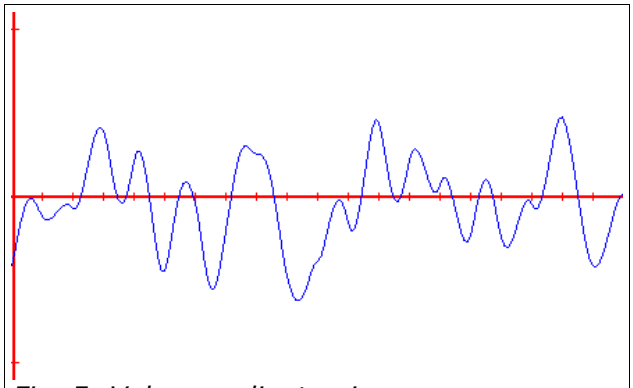


Fig. 5: Value-gradient noise

Combining Noise.

The noise functions that we made look a bit like badly mangled sinusoid. Every integer point has a new random value, and the whole noise function tends to change direction once per integer point. We can say that our noise has a *frequency* of 1. Frequency of a noise function is a very important concept. It determines how fast the function changes, - in other words, how noisy the noise is. Most usages of noise functions involve combining different frequencies together to achieve desired effect.

CoherentNoise library allows for literally unlimited possibilities, but first let's look at some simple ideas. In most basic form, noise function can be used to distort some pattern - that may be itself generated procedurally or made by hand. We can add noise directly to the pattern, or we can use noise values to

perturb the pattern. Perturbation of function is achieved by adding noise value to function inputs, like this: $p(x)=f(x+noise(x))$. Figure 6 shows the effect of addition and perturbation on a 2D pattern.

Fig. 6: Effects of addition and perturbation

This effect is nice, but not very impressive. In fact, natural phenomena usually have more varied frequencies. Most often, a natural pattern has irregularities on different scales. Consider a hilly terrain, for example. It has a "random" distribution of large hills, that are themselves irregular, with smaller pits and ridges. To achieve something similar with noise, we can add together several noise functions, with different frequencies and different weights:

$$f(x)=noise_1(x)+\frac{1}{2}noise_2(x)+\frac{1}{4}noise_4(x)+\ldots$$

Here $noise_1$ function has a frequency of 1, $noise_2$ - frequency of 2, $noise_4$ - frequency of 4 and so on. This kind of noise is called *fractal noise*. A sample of fractal noise is shown in figure 7.

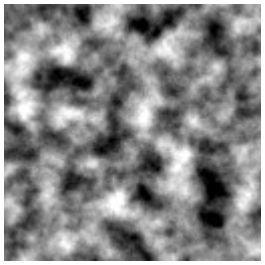


Fig. 7: Fractal noise

True fractal noise has an infinite number of terms in its formula. Of course, this is not possible in practice; actual fractal noise functions in CoherentNoise library are *band-limited*. This simply means that they only add together a small number of noise functions, up to a certain cutoff frequency.

CoherentNoise library.

CoherentNoise API is built around the concept of Generator. A generator is simply a function that returns a float number given three float coordinates. It can represent a non-random function, or a noise function like value or gradient noise. It can also represent some mix of these. Many generators also work as modifiers - i.e. they somehow change values of another "source" generator.

With use of operator overloading, generators can be added, subtracted, multiplied together. Also, any float number can be converted to a generator that returns this number irrespective of coordinates (a constant generator). This allows to write natural-looking code, like this:

```
var vgnoise = 0.5f*(new GradientNoise() + new ValueNoise());
```

The most basic generators are `ValueNoise`, `GradientNoise` and `Constant`. The first two implement noise function that were discussed earlier; the last is simply a constant function. Other generators are divided into categories:

- Patterns. These are generators that create regular patterns. They are not noisy per se, but are often used as a base function that is then changed by noise.

- Fractal noise. Fractal generators combine several noise functions with increasing frequencies. Pink noise, also sometimes called Perlin noise, falls into this category, as well as a couple of others. Abstract class `FractalNoiseBase` can be subclassed to create your own crazy noise functions.

- Voronoi. Voronoi diagrams are not exactly noise functions, but they're still “random”. These generators create cell-like patterns. Again, abstract base class can be subclassed to create your own diagrams.

- Modification. These generators do not generate values of their own, but rather modify another generator. They can be used to apply a function to noise value, map it to a curve, etc.

- Displacement. A counterpart to modification, displacement generators rely on a source too. But they modify not the noise value, but input coordinates. These generators can be used to rotate of stretch noise patterns. Or a noise value can be added to input coordinates, producing turbulent, swirly patterns.

- Combination. These generators combine two or more source functions. They can be used to add, multiply, select minimum or maximum value of two functions, or blend two patterns together using third as a weight.

Some generators, most notably value and gradient noise, have a 2D variant. It is a marginally (or in case of Voronoi, significantly) faster version of a generator that ignores the third coordinate. These generators can be used if all you care about is 2D, not 3D patterns.

An important thing to remember is the extent of noise functions. Value and gradient noise, as well as pattern generators, are guaranteed to produce values between -1 and 1. Most other generators try to do the same, although they cannot actually guarantee it - it's hard to be sure of anything when dealing with random numbers. Thus, when using CoherentNoise it pays to keep your noise' values roughly within this range - this will ensure all modification and combination generators work as intended. For example, when adding two generators together, the resulting function will probably extend form -2 to 2. To prevent that, just divide it by 2!

Chapter II

Noise generators

`Generator base class`

The base class for all generators. It has one abstract method:

```
public abstract float GetValue(float x,float y,float z);
```

The method returns noise function value at given coordinates. Although it is not explicitly required, all code treats this method as a function in mathematical sense. That is, this method should always return the same value given the same inputs.

`Generator` class also implements operator overloading:

```
public static Generator operator+(Generator g1, Generator g2);
public static Generator operator+(Generator g, float f);
public static Generator operator-(Generator g1, Generator g2);
public static Generator operator-(Generator g, float f);
public static Generator operator*(Generator g1, Generator g2);
public static Generator operator*(Generator g, float f);
public static Generator operator/(Generator g1, Generator g2);
public static Generator operator/(Generator g, float f);
```

And cast operator:

```
public static implicit operator Generator(float f);
```

These allow arithmetical operations on generators and numbers. Numbers are converted to `Constant` generator, that always returns constant value. Arithmetical operations return generators that apply the operation to their sources' values.

In addition to these, there's a number of extension methods defined for `Generator` class. They are used to create new generators in a more readable way. These methods are:

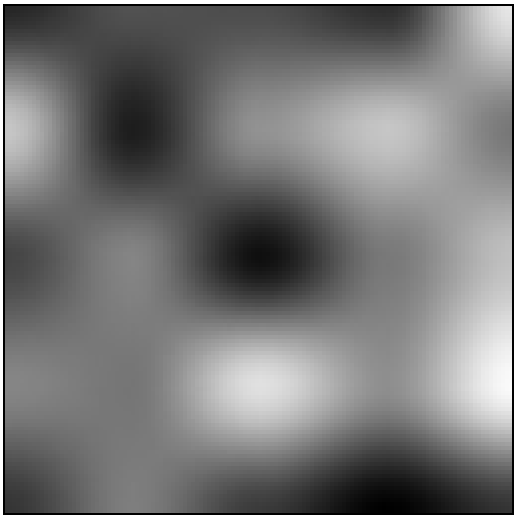
```
public static Generator Scale(this Generator source, float x, float y, float z)

public static Generator Translate(this Generator source, float x, float y, float z)

public static Generator Rotate(this Generator source, float x, float y, float z)

public static Generator Turbulence(this Generator source, float frequency, float power)

public static Generator Blend(this Generator source, Generator other, Generator weight)

public static Generator Modify(this Generator source, Func<float, float> modifier)
```

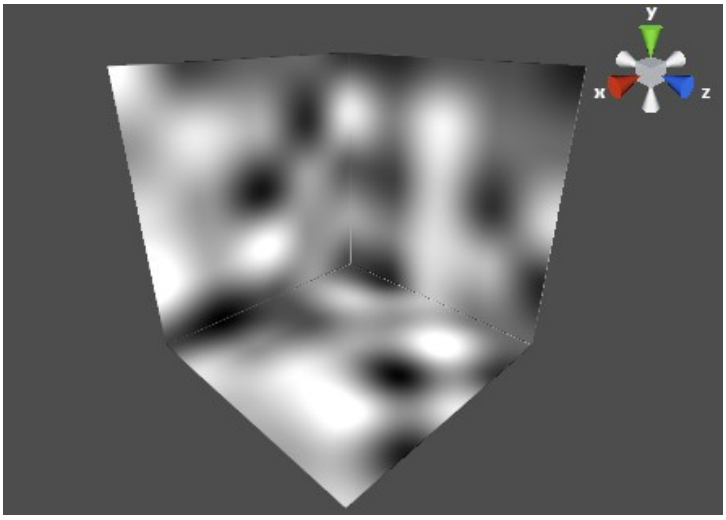
```
public static Generator Curve(this Generator source, AnimationCurve curve)

public static Generator Binarize(this Generator source, float treshold)

public static Generator Bias(this Generator source, float b)

public static Generator Gain(this Generator source, float g)

public static Generator ScaleShift(this Generator source, float a, float b)
```

All these methods (except `ScaleShift`) simply create a new generator of the same name. Read documentation for corresponding classes to understand what they do.

The `ScaleShift` method is the only exception. It returns a generator that is a linear transform of source. That is, `noise.ScaleShift(a, b)` is equivalent to `a*noise + b`. This method is useful to change noise range from [0,1] to [-1,1] and vice versa.

ValueNoise





This generator creates value noise, as described in Chapter I of this manual.

`ValueNoise` defines two constructors:

```
public ValueNoise(int seed);
public ValueNoise(int seed, SCurve sCurve)
```

`seed` value is used to seed the underlying random number generator. Two generators with the same seed will produce same results, which might be useful if you want to reproduce the generated function.

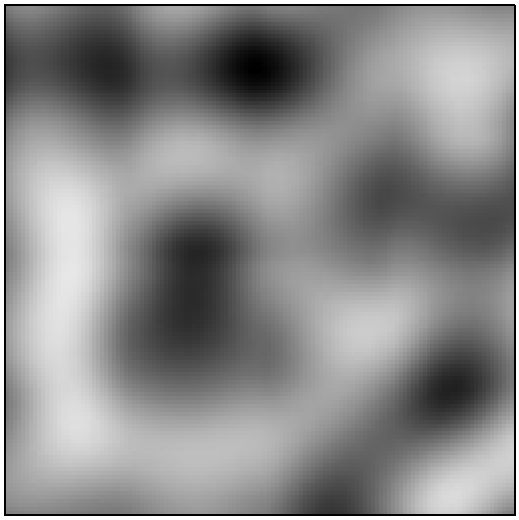
`sCurve` defines interpolation algorithm used. See `SCurve` class documentation for details. First constructor uses `Scurve.Default` as an interpolation, which defaults to cubic.

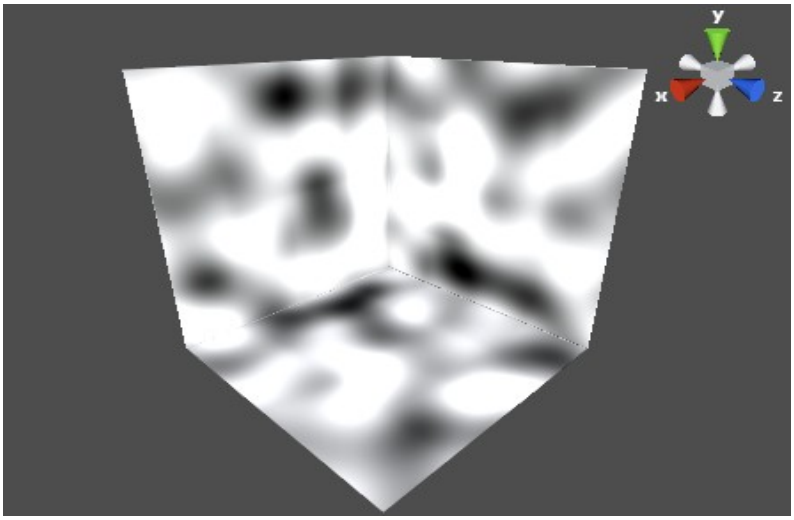
`ValueNoise` defines one property:

```
public int Period {get;set;}
```

`Period` is used for repeating noise patterns. Sometimes, especially when building seamless textures, it is desirable to have "random" noise that repeats exactly every N units. When `Period` value is not zero, the generator would behave just like this: it will repeat generated pattern every `Period` units. `Period` is integer, not float, because it actually sets period of the underlying integer-point lattice. When `Period` is set to zero (this is the default), the generator will not repeat itself. Negative values of `Period` are invalid and would throw an exception.

GradientNoise





This generator creates gradient noise, as described in Chapter I of this manual.

`GradientNoise` defines two constructors:

```
public GradientNoise(int seed);
public GradientNoise(int seed, SCurve sCurve)
```

`seed` value is used to seed the underlying random number generator. Two generators with the same seed will produce same results, which might be useful if you want to reproduce the generated function.

`sCurve` defines interpolation algorithm used. See `SCurve` class documentation for details. First constructor uses `Scurve.Default` as an interpolation, which defaults to cubic.

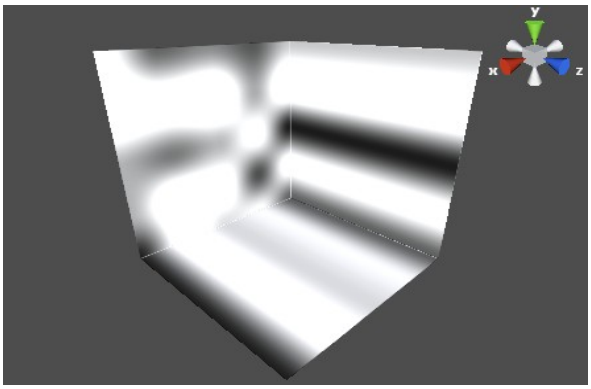
`GradientNoise` defines one property:

```
public int Period {get;set;}
```

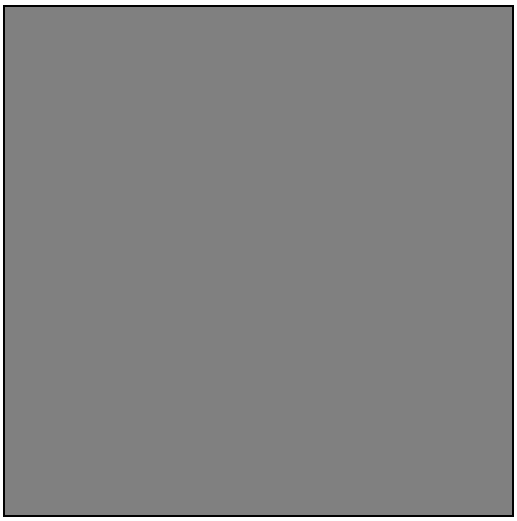
`Period` is used for repeating noise patterns. Sometimes, especially when building seamless textures, it is desirable to have "random" noise that repeats exactly every N units. When `Period` value is not zero, the generator would behave just like this: it will repeat generated pattern every `Period` units. `Period` is integer, not float, because it actually sets period of the underlying integer-point lattice. When `Period` is set to zero (this is the default), the generator will not repeat itself. Negative values of `Period` are invalid and would throw an exception.

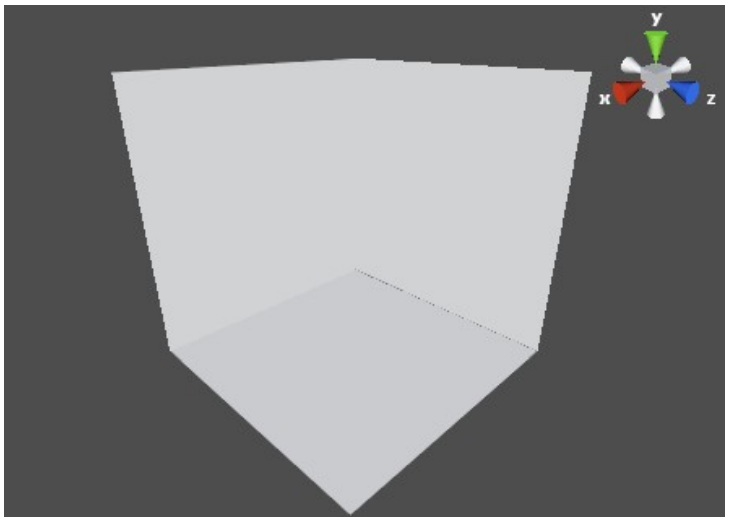
ValueNoise2D and GradientNoise2D

These are variants of `ValueNoise` and `GradientNoise` that ignore Z coordinate. Use these generators if you only need 2D patterns, as 2D generators are faster.



Constant





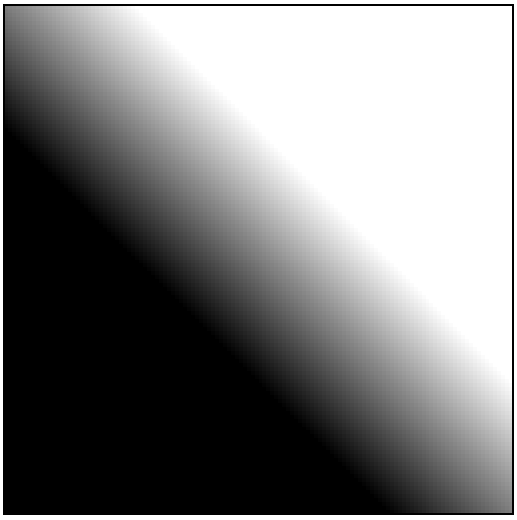
This is probably the simplest generator. It ignores the input coordinates and always returns one value. Constant is almost never used directly, as a float number can be implicitly cast to it.

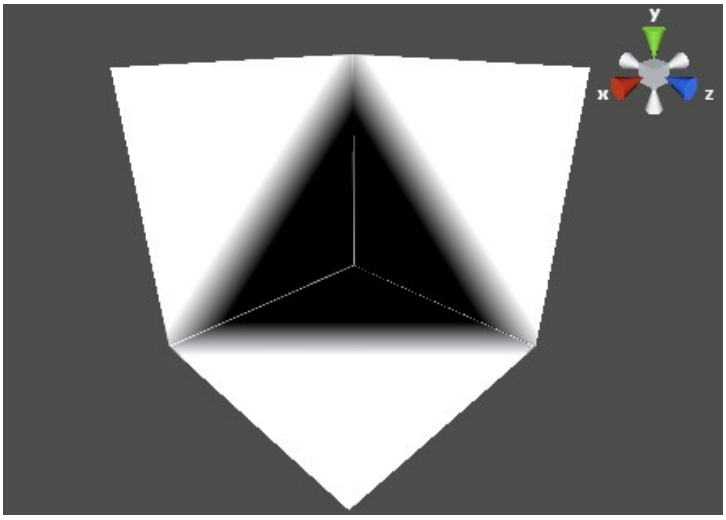
`Constant` has only one constructor:

```
public Constant(float value);
```

The `value` parameter is the value that the generator would return.

Function





The `Function` generator wraps a delegate that accepts three floats and returns a float. Usually, it is used with a lambda expression to create some pattern.

The generator has one constructor:

```
public Function(Func<float,float,float,float> func);
```

For example, a function that is shown here is created with the following code:

```
var f = new Function((x,y,z) => x+y+z-4);
```

Cache

The `Cache` generator does not generate anything. It simply wraps another generator, adding a simple cache: so that when asked for the same value twice, it does not recalculate it. This is useful when the same generator is used more than once. Note, however, that `Cache` only caches one most recent value. This means that caching will only work when cached generator is evaluated at the same coordinates. Using it as a source for displacement or fractal noise generators will usually break the cache.

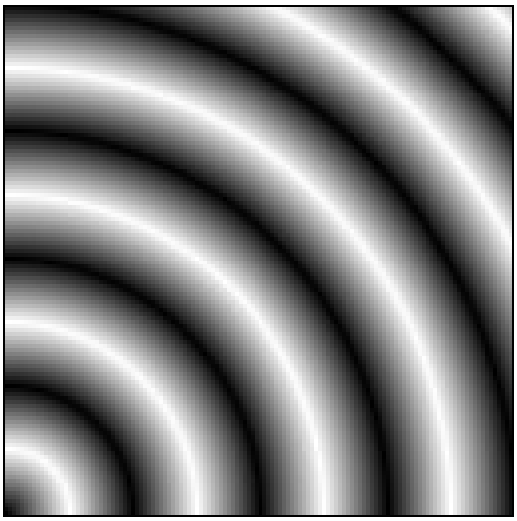
`Cache` has one constructor:

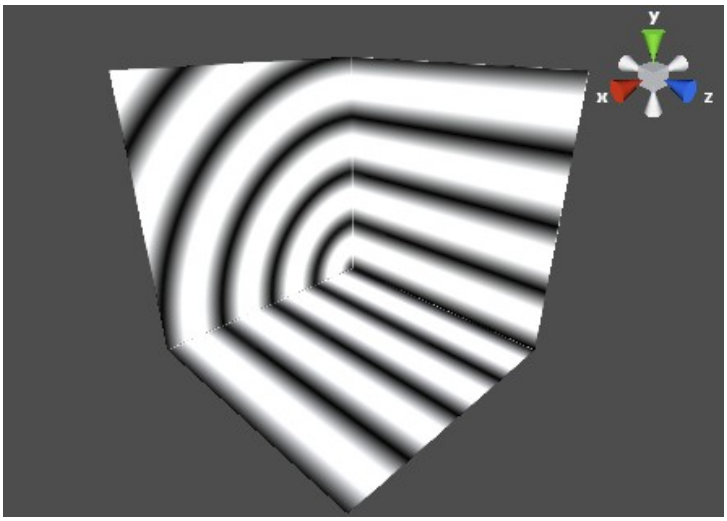
```
public Cache(Generator source);
```

`source` is the generator to be cached. It cannot be `null`.

Patterns

Cylinders





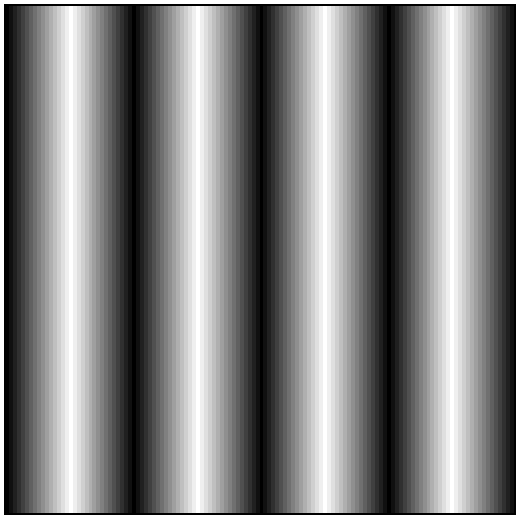
`Cylinders` generator creates a pattern of concentric cylinders, parallel to Z axis. Resulting value alternates (linearly) from -1 to 1, starting with -1 at Z axis and rising to 1 at supplied radius.

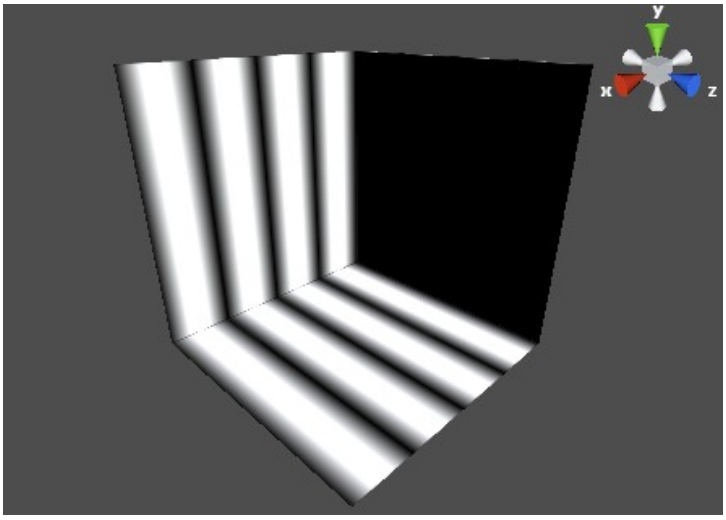
The generator has one constructor:

```
public Cylinders(float radius);
```

Where `radius` is the radius of cylinders. The constructor throws an exception if `radius<=0`.

Planes





This generator creates a pattern of planes, parallel to YZ plane. Resulting value

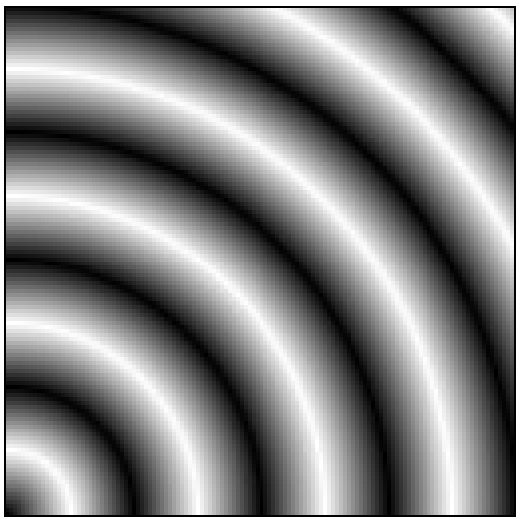
goes from -1 on the YZ plane (i.e. when X=0) to 1 at supplied distance from it, then back to -1 etc.

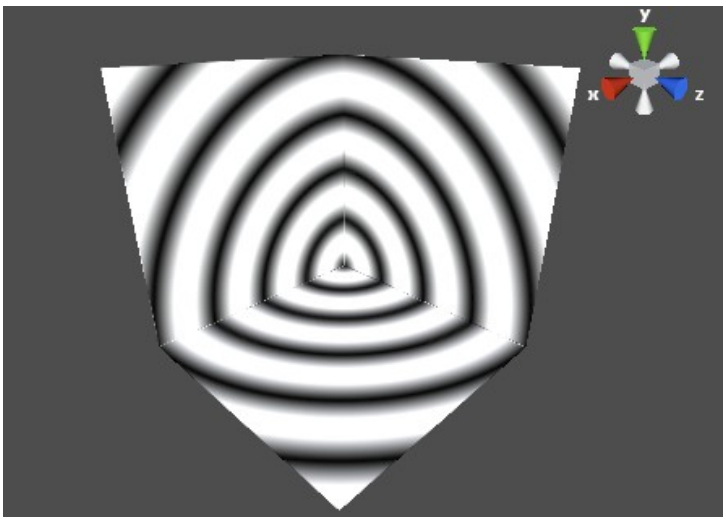
`Planes` generator has one constructor:

```
public Planes(float step);
```

Where `step` is the distance between -1 and 1 values. The constructor throws an exception if `step<=0`.

Spheres





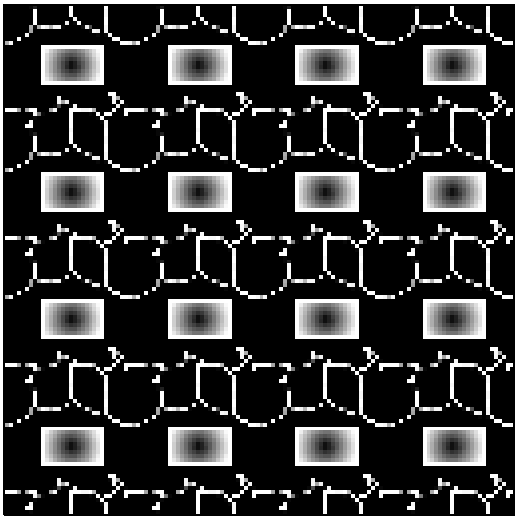
This generator creates a pattern of concentric spheres, centered at origin. Resulting value goes from -1 at the origin to 1 at supplied radius, then back to -1 etc.

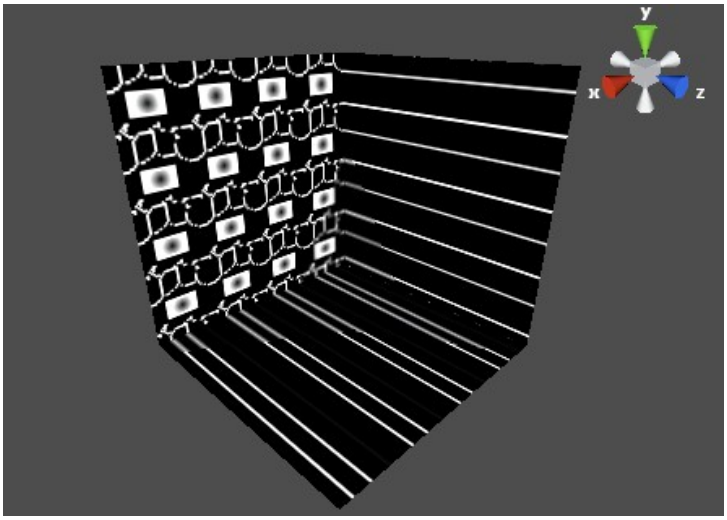
`Spheres` generator has one constructor:

```
public Spheres(float radius);
```

Where `radius` is the distance between -1 and 1 values. The constructor throws an exception if `radius<=0`.

TexturePattern





The `TexturePattern` generator is a bridge between a hand-drawn texture and noise generation. It takes a monochrome texture as input and returns color intensity as noise value. The generator uses a brain-dead nearest-neighbour algorithm to find out color at a given point, which can lead to visual artifacts when texture resolution does not match final resolution. You can actually see these artifacts in the image above, as 128x128 texture is used for a 32x32 region in the resulting image.

Supplied texture is mapped to input coordinates from 0 to 1. Coordinates outside of that range may be either clamped to [0,1], or set to repeat.

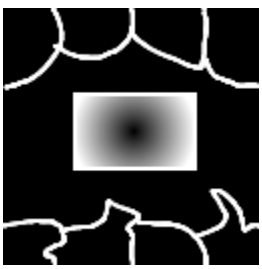
Since texture is fundamentally 2D, `TexturePattern` generator ignores Z coordinates.

The generator has one constructor:

```
public TexturePattern(Texture2D texture,TextureWrapMode wrapMode)
```

Where `texture` is the texture to use, and `wrapMode` defines behavior when input coordinates are outside [0,1] range.

The images above were made with `wrapMode=TextureWrapMode.Repeat` and the following `texture`:



Fractal

FractalNoiseBase

`FractalNoiseBase` class is the base class for all fractal generators. Fractal generators take a source noise generator and combine together values sampled at different frequencies. Exact combination algorithm is not implemented in the base class. By overriding abstract method

```
protected abstract float CombineOctave(int curOctave, float signal, float value)
```

derived classes can implement this algorithm.

The class has two constructors:

```
protected FractalNoiseBase(Generator source)
protected FractalNoiseBase(int seed)
```

First constructor, taking the source parameter initializes source noise generator. Second constructor uses seed value to initialize source with a `GradientNoise` generator.

The `CombineOctave` abstract method is the heart of this class. Internally, it is called for each sampling of the source generator. These samples are called *octaves.* The generation starts with octave 0. Source is sampled at current point, and current noise value is obtained from a call to `CombineOctave`. Then, point coordinates are scaled (the coordinates are multiplied by `Lacunarity`), thus increasing noise frequency, then rotated to break up regular patterns. Source is sampled at this new point, and new value is obtained by calling `CombineOctave` again. This step repeats an `OctaveCount` number of times, and the final value is returned.

`CombineOctave` parameters mean:

`curOctave`: current step number. Starts at 0 and goes to `OctaveCount-1`.

`signal`: value of the source generator at this step.

`value`: resulting value from the previous step.

`CombineOctave` must return final value for the step. Normally, the returned value is the same whether it it the final step or not (provided all other parameters are equal). That is, when `OctaveCount` is set to 5, `CombineOctave` shall return the same value for `curOctave==4` as when `OctaveCount` is 6 or more.

`FractalNoiseBase` has the following properties:

```
public float Frequency {get;set;}
public float Lacunarity {get;set;}
```

```
public int OctaveCount {get;set;}
```

`Frequency` is the initial sampling frequency. It is equivalent to stretching the source by 1/`Frequency`. `Frequency` must be greater than zero.

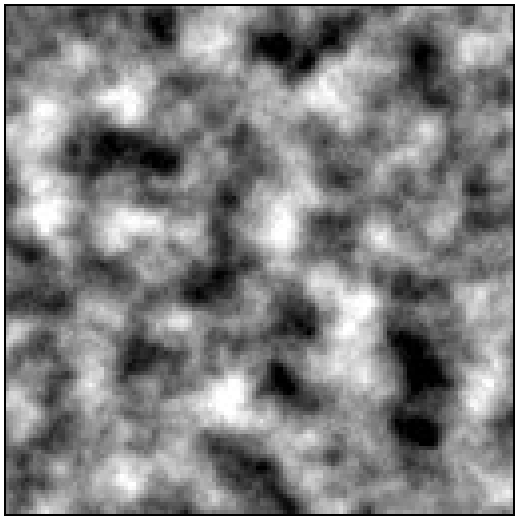
`Lacunarity` is the change of frequency between octaves. Each consequent octave multiplies coordinates by `Lacunarity` value. The default value, 2.17, is good for most purposes. Note that it is not set to exactly 2 – small fraction is added to further break up any regularities. `Lacunarity` must be more than 1.

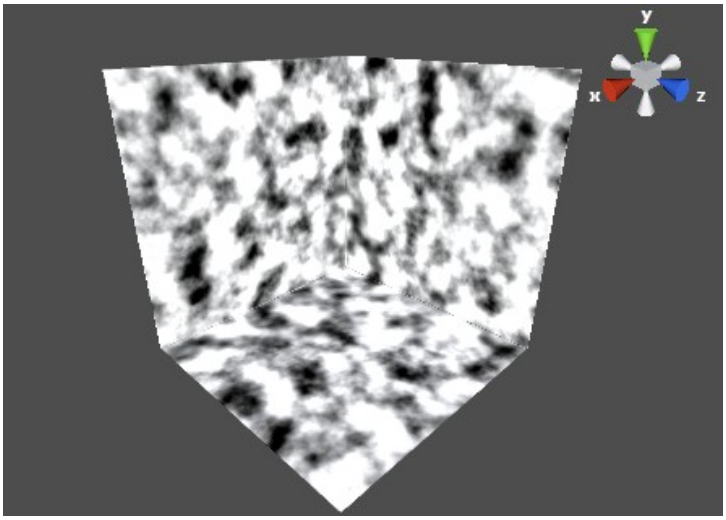
`OctaveCount` is the number of steps, or terms in fractal formula. `OctaveCount` must be at least 2. Default value is 6 steps.

The `OnParamsChanged` virtual method is called whenever any of these properties change. It does not do anything in the base class, but provides an extension point for derived classes. They can implement any precalculations here.

```
protected virtual void OnParamsChanged()
```

PinkNoise





`PinkNoise` is a simple variant of fractal noise. It adds together all signals, weighting them with inverse frequency – so that higher frequencies have less weight. Another name for this algorithm is *Fractional Brownian Motion*.

`PinkNoise` values are meant to fall between -1 and 1, but this is not strictly guaranteed. Mean value of `PinkNoise` is 0.

`PinkNoise` defines two constructors:

```
public PinkNoise(Generator source)
public PinkNoise(int seed)
```

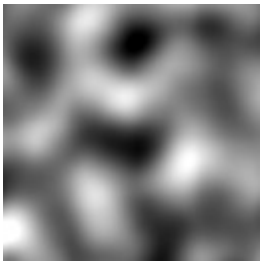
These work exactly like in the base class. First constructor, taking the source parameter initializes source noise generator. Second constructor uses seed value to initialize source with a `GradientNoise` generator.

In addition to base class' properties, `PinkNoise` defines one property:

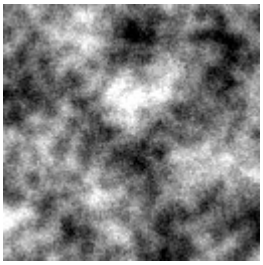
```
public float Persistence {get;set;}
```

`Persistence` defines weighting of individual frequencies. Each step, the signal is multiplied by `Persistence` value. Higher `Persistence` means more weight is given to higher frequencies, and vice-versa. `Persistence` values higher than 1 are possible, but will probably not produce useful results. Default value is 0.5.

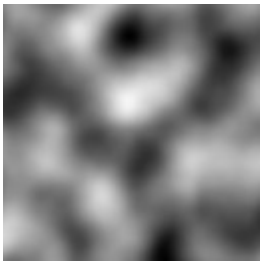
The following pictures show the effect of different settings on resulting noise. Compare them with picture above, that shows noise with all default values.



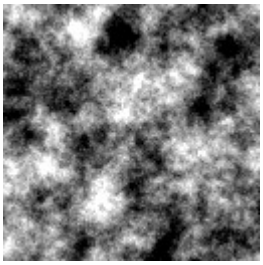
Lacunarity = 1.2



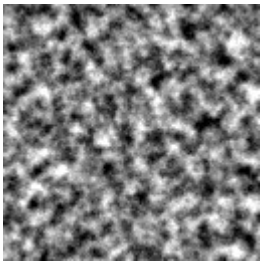
Lacunarity = 4.3



Persistence = 0.17

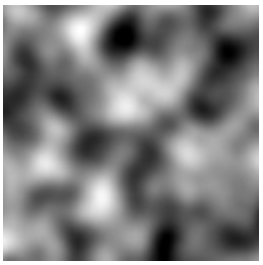


Persistence = 0.71

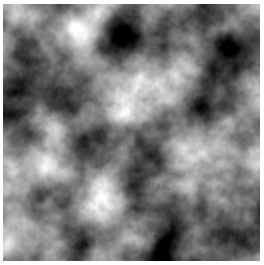


Frequency = 4

Frequency = 0.2

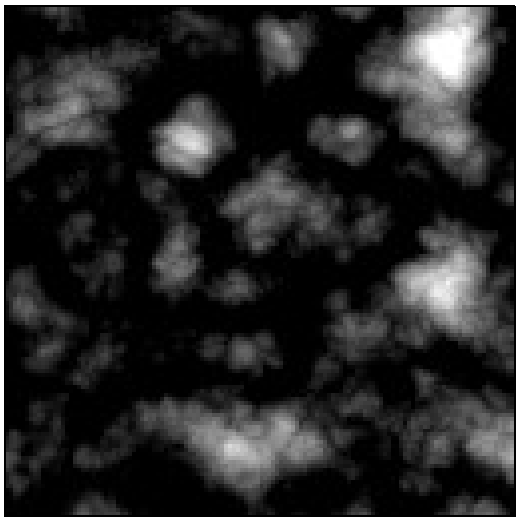


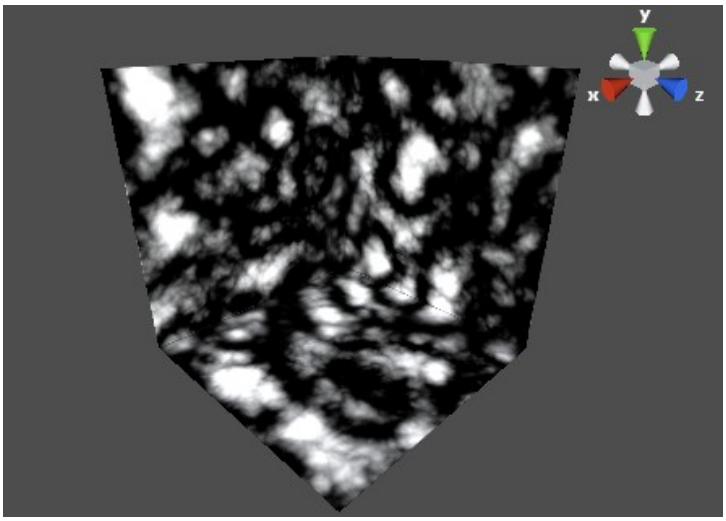
OctaveCount = 2



OctaveCount = 12

BillowNoise





Billow noise is a variant of pink noise, that adds together absolute values of samples. It produces cloud-like shapes as a result. `BillowNoise` values generally fall into [-1,1] range, and mean value is 0.

`BillowNoise` defines two constructors:

```
public BillowNoise(Generator source)
public BillowNoise(int seed)
```

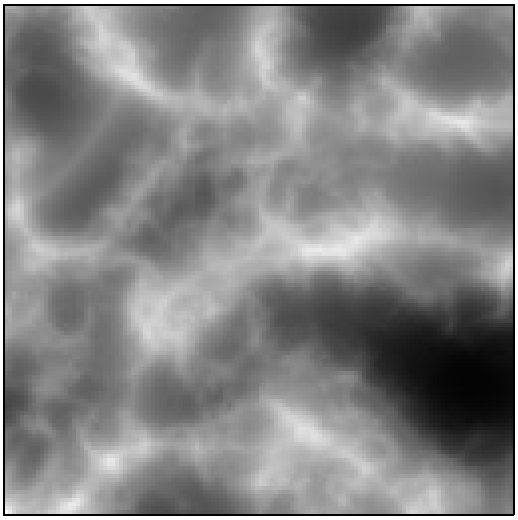
These work exactly like in the base class. First constructor, taking the source parameter initializes source noise generator. Second constructor uses seed value to initialize source with a `GradientNoise` generator.

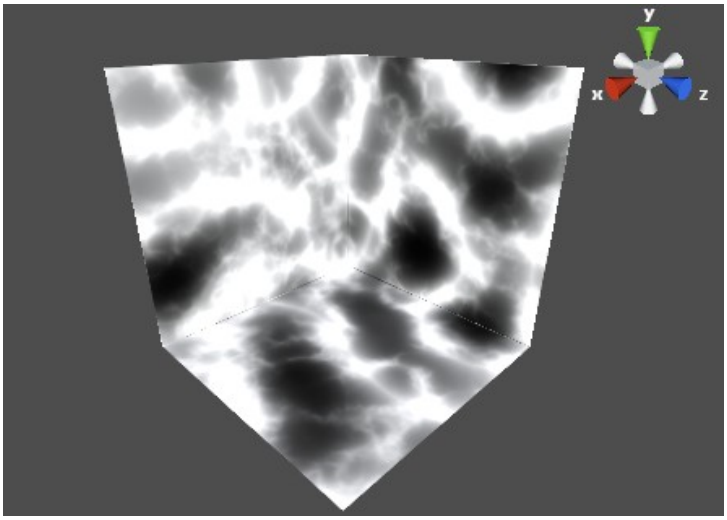
In addition to base class' properties, `BillowNoise` defines one property:

```
public float Persistence {get;set;}
```

`Persistence` defines weighting of individual frequencies. Each step, the signal is multiplied by `Persistence` value. Higher `Persistence` means more weight is given to higher frequencies, and vice-versa. `Persistence` values higher than 1 are possible, but will probably not produce useful results. Default value is 0.5.

RidgeNoise





`RidgeNoise` is a so-called *multifractal* noise. Compared to simple fractals, multifractals have different roughness in different places. To achieve this, `RidgeNoise` derives weight of a given frequency from values of lower-frequency samples. Also, an offset is added at each step, and a gain value is multiplied in, creating a sort of feedback loop. This ensures that rough areas tend to become rougher, and smooth areas to become smoother. The resulting function is really great for terrain generation, creating both smooth valleys and rough mountain ridges.

`RidgeNoise` defines two constructors:

```
public RidgeNoise(Generator source)
```

```
public RidgeNoise(int seed)
```

These work exactly like in the base class. First constructor, taking the source parameter initializes source noise generator. Second constructor uses seed value to initialize source with a `GradientNoise` generator.

In addition to base class' properties, `RidgeNoise` defines the following ones:

```
public float Exponent {get;set;}
```

```
public float Offset {get;set;}
```

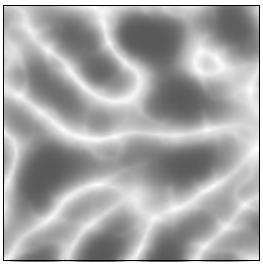
```
public float Gain {get;set;}
```

`Exponent` controls weights given to different frequencies (like `Persistence` in `PinkNoise`). Base weight of a signal is determined by raising frequency to the power of -Exponent. Thus, higher `Exponent` makes for less high-frequency contribution. Default value is 1.

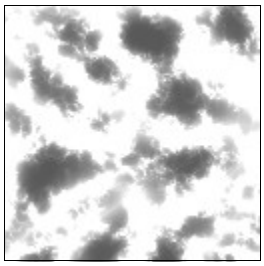
`Offset` is added to the signal at each step. Higher `Offset` means more rough results (more ridges). Default value is 1.

`Gain` is the "feedback" factor that scales the lower-frequency contribution in high-frequency samples. Higher `Gain` values mean noisier ridges. Default value is 2.

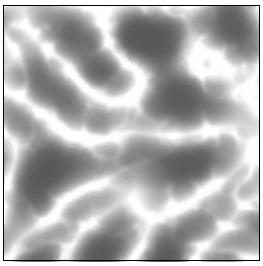
The following pictures show the effect of different settings on resulting noise. Compare them with picture above, that shows noise with all default values.



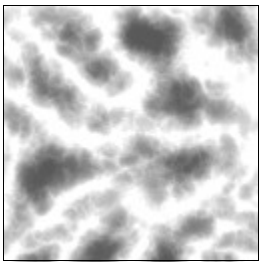
Exponent=2



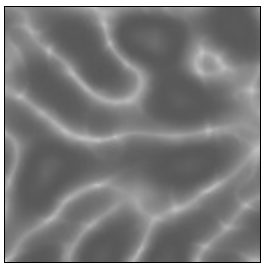
Exponent=0.4



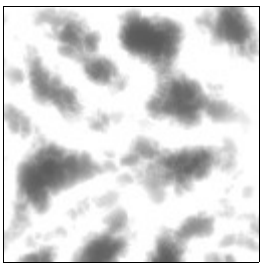
Gain=1.2



Gain=4



Offset=0.7



Offset=1.1

Voronoi

VoronoiDiagramBase and VoronoiDiagramBase2D

Voronoi diagrams are used to create cell-like or network-like patterns. A diagram is based on an (infinite) number of *control points*, that are randomly distributed in space. For any point in space, a set of several closest control points can be determined. Distances to them are used to calculate final "noise" value. As actually comparing an infinite number of points is impractical, CoherentNoise uses an optimized version, where control points are obtained by shifting points with integer coordinates. This means that every unit-sized cube will have exactly one control point in it. Such distribution is not completely random, but it is enough for most practical applications.

Base classes for Voronoi diagrams determine distances to three closest control points, but delegate actual value computation to an abstract method, `GetResult`. This allows to make your own diagrams by combining these distances in creative ways.

All Voronoi diagrams have a 2D variant. These variants are exactly the same as their 3D counterparts, except they ignore the Z coordinate. This makes them considerably faster (as less control points must be considered).

`VoronoiDiagramBase` defines one constructor:

```
protected VoronoiDiagramBase(int seed)
```

`seed` value is used to seed the internal noise generator that computes distribution of control points.

`VoronoiDiagramBase` defines two properties:

```
public int Period {get;set;}

public float Frequency {get;set;}
```

`Period` is used for repeating noise patterns. Sometimes, especially when building seamless textures, it is desirable to have "random" noise that repeats exactly every N units. When `Period` value is not zero, the generator would behave just like this: it will repeat generated pattern every `Period` units. `Period` is integer, not float, because it actually sets period of the underlying integer-point lattice. When `Period` is set to zero (this is the default), the generator will not repeat itself. Negative values of `Period` are invalid and would throw an exception.

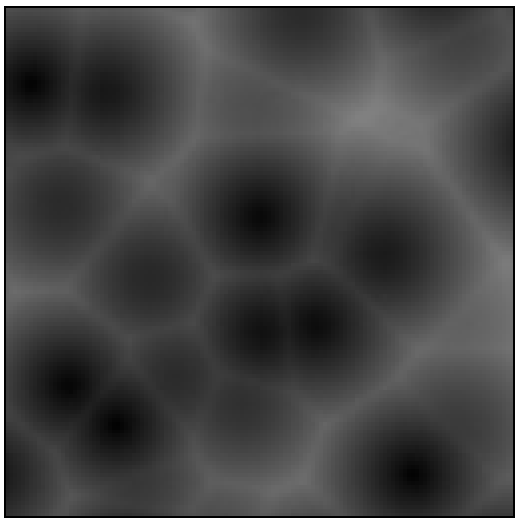
`Frequency` determines scaling of control points. All coordinates are multiplied by `Frequency`, thus higher `Frequency` means smaller cells. Another way to think of `Frequency` is as of average distance between control points. Default value is 1.

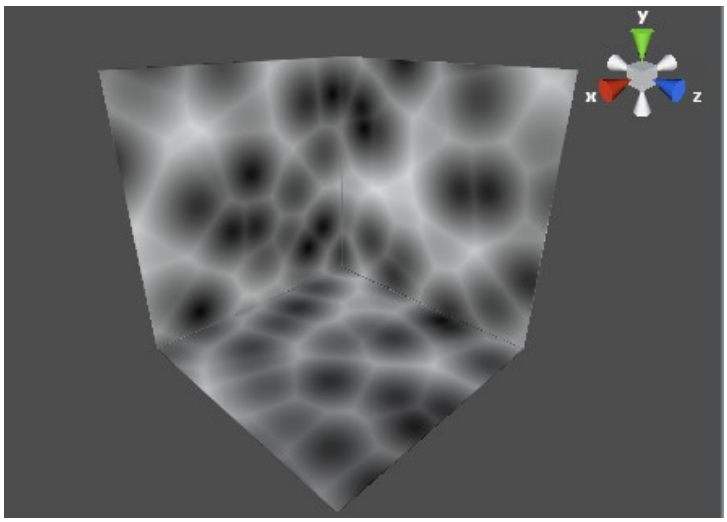
An abstract method `GetResult`

```
protected abstract float GetResult(float min1,float min2,float min3);
```

is used to calculate final noise value. Its parameters are distances, respectively, to the closest control point, second-closest and third-closest. Note that these distances are already in frequency-adjusted coordinates. This means that no matter the value of `Frequency`, distances in `GetResult` work as if `Frequency` was equal to 1.

`VoronoiPits` and `VoronoiPits2D`





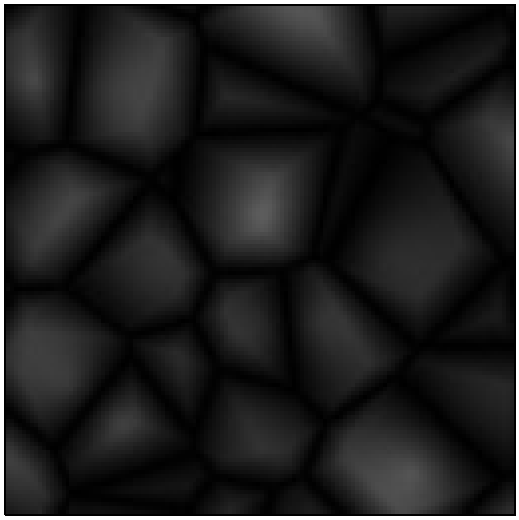
This is the simplest Voronoi diagram, that simply returns distance to closest control point as a result. This creates "pits" with value of 0 at control points, and higher values in between.

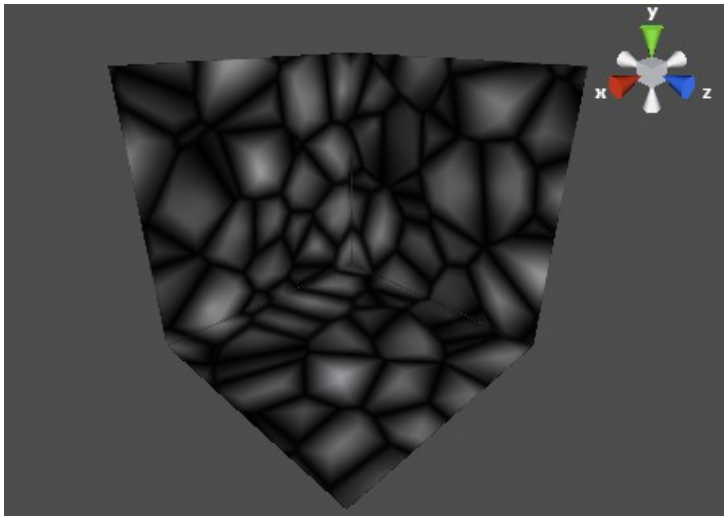
`VoronoiPits` defines one constructor:

```
public VoronoiPits(int seed)
```

`seed` value is used to seed the internal noise generator that computes distribution of control points.

VoronoiValleys and VoronoiValleys2D





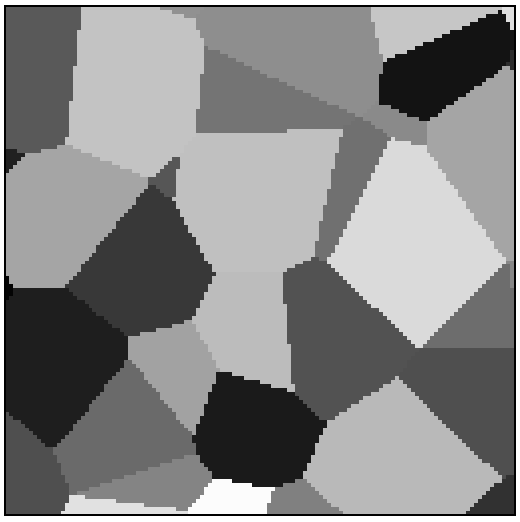
This Voronoi diagram returns difference in distance to two closest control points. The resulting patter looks like "valleys" with value 0 at the bottom, and highest values at control points.

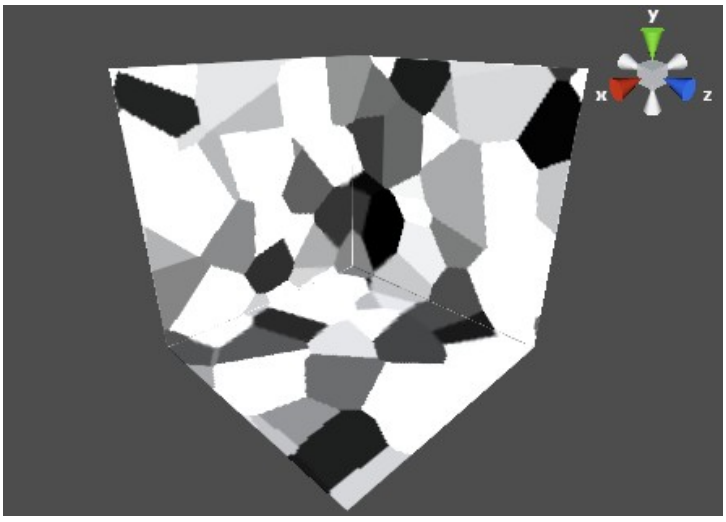
`VoronoiValleys` defines one constructor:

```
public VoronoiValleys(int seed)
```

`seed` value is used to seed the internal noise generator that computes distribution of control points.

VoronoiCells and VoronoiCells2D





Voronoi cells is a special kind of Voronoi diagram. Instead of using distances to obtain resulting value, cell diagram uses the closest control point itself (or rather, its coordinates). This generator uses a user-supplied function, that transforms integer control point coordinates into final value. As an example, this function can use the coordinates to obtain a pseudo-random number.

`VoronoiCells` defines one constructor:

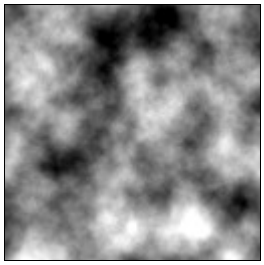
```
public VoronoiCells(int seed, Func<int,int,int,float> cellValueSource)
```

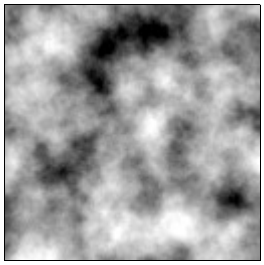
`seed` value is used to seed the internal noise generator that computes distribution of control points.

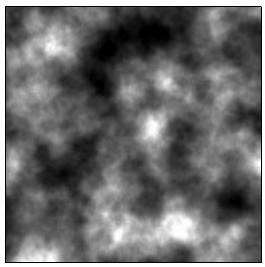
`cellValueSource` is the function that maps cell coordinates to cell value.

Modification

Bias


Source


Bias=0.5


Bias=-0.5

`Bias` generator is used to shift noise mean value, while leaving it within [-1,1] interval. It maps noise value to a power-like curve (the original *bias* function, devised by Perlin, uses power curve, but this generator uses a faster approximation). `Bias` is symmetric, meaning that applying two `Bias` generators one after another with opposite bias values will result in unmodified source noise.

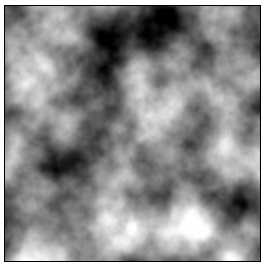
`Bias` clamps source noise to [-1,1] interval, as values outside of this interval may cause division by zero errors.

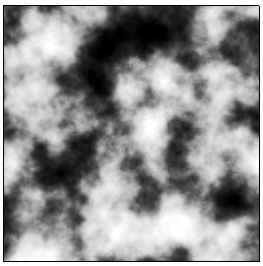
`Bias` has one constructor:

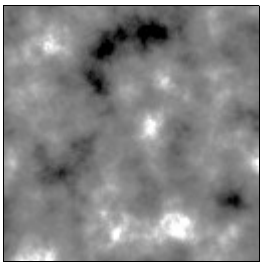
```
public Bias(Generator source, float bias)
```

`source` is the source generator, `bias` is the bias value. Resulting noise will be equal to `bias` value in point where it was 0. `bias` can only be between -1 and 1, non-inclusive; invalid values will cause an `ArgumentException`.

Gain


Source


Gain=0.5


Gain=-0.5

`Gain` generator is used to "sharpen" noise, shifting high values even higher, and low values even lower. Resulting noise stays in [-1,1] interval, and values of 0 remain unchanged. `Gain` is symmetric, meaning that applying two `gain` generators one after another with opposite gain values will result in unmodified source noise. Internally, it applies two bias functions – one for positive values and one for negative.

`Gain` clamps source noise to [-1,1] interval, as values outside of this interval

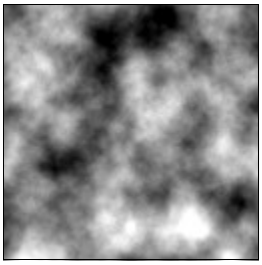
may cause division by zero errors.

`Gain` has one constructor:

```
public Gain(Generator source, float gain)
```

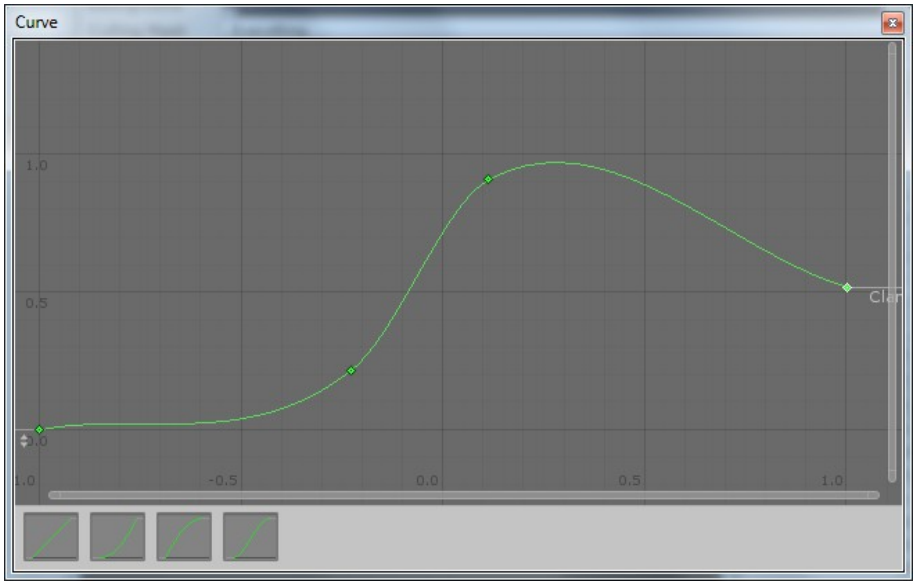
`source` is the source generator, `gain` is the gain value. Resulting noise will be shifted exactly by `gain/2` value in points where it was equal to 0.5, and by `-gain/2` in points where it was equal to -0.5. `gain` can only be between -1 and 1, non-inclusive; invalid values will cause an `ArgumentException`.

Curve



Source

Result



Curve used

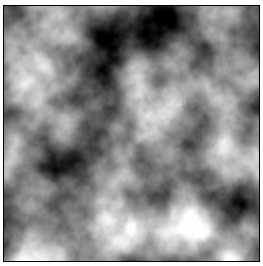
`Curve` generator maps a source to an `AnimationCurve` that can be edited inside Unity3d. This allows for great flexibility and is actually quite fast. Note that by default, curves in Unity3d only stretch from 0 to 1, and curves for use with CoherentNoise should probably stretch from -1 to 1.

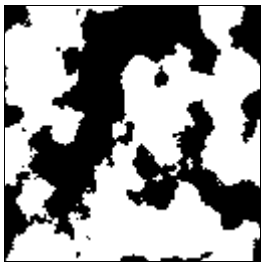
`Curve` generator has one constructor:

```
public Curve(Generator source, AnimationCurve curve)
```

`source` is the source generator, and `curve` is the curve object to use.

Binarize


Source


Result

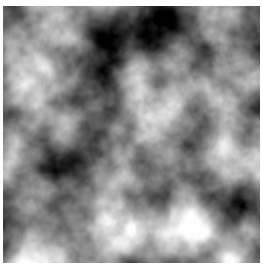
`Binarize` generator is an extreme sharpener, that turns noise into function with only two values: -1 and 1. All values less than threshold become -1, all greater become 1.

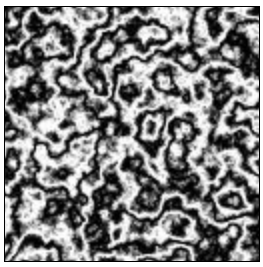
`Binarize` has one constructor:

```
public Binarize(Generator source, float threshold)
```

`source` is the source generator, `threshold` is the threshold value.

Modify


Source


*Modified with Sin(10*x)*

`Modify` generator is the universal modification: it simply applies a user-supplied function to source value.

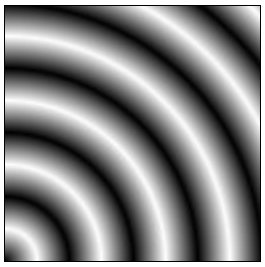
`Modify` has one constructor:

```
public Modify(Generator source, Func<float, float> modifier)
```

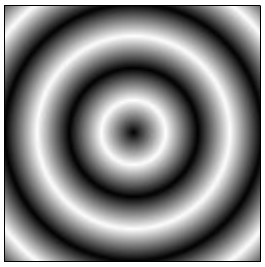
`source` is the source generator, `modifier` is the modification function.

Displacement

Translate



Source



Translated by (-2,-2,0)

`Translate` generator moves its source around. This is useful when working with patterns (e.g. `Spheres` pattern is always centered at (0,0,0). With `Translate`, you can move the center anywhere you like).

`Translate` has two constructors:

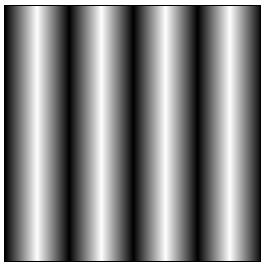
```
public Translate(Generator source, Vector3 v)
public Translate(Generator source, float x, float y, float z)
```

`source` is the source generator.

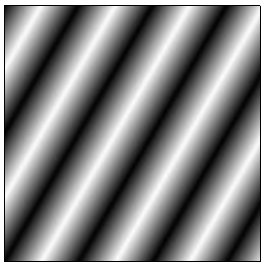
`v` is the translation vector, and `x,y,z` are its components.

Note that translation is applied to *input coordinates,* not the noise itself. This means that the value that was at (0,0,0) will be translated to (-x,-y,-z)!

Rotate



Source



Rotated by 30°

`Rotate` generator rotates its source around origin. This is useful when working with patterns (e.g. `Planes` pattern is always parallel to YZ plane. With `Rotate`, you can incline it however you like).

Rotate has two constructors:

```
public Rotate(Generator source, Quaternion rotation)
public Rotate(Generator source, float angleX, float angleY, float angleZ)
```

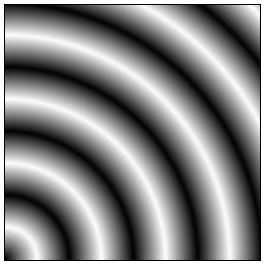
`source` is the source generator.

`rotation` is the desired rotation

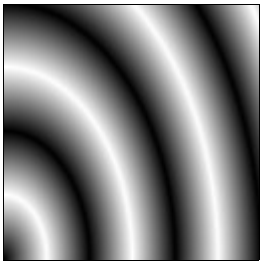
`angleX`, `angleY` and `angleZ` are Euler angles that define desired rotation.

Note that rotation is applied to *input coordinates*, not the noise itself. This means that noise pattern will rotate in opposite direction!

Scale



Source



Scaled by (0.75,0.5,1)

`Scale` generator scales its source, in effect changing its frequency.

`Scale` has two constructors:

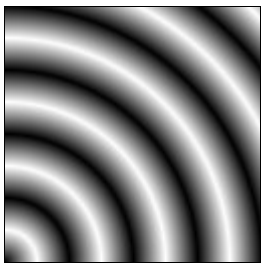
```
public Scale(Generator source, Vector3 v)
public Scale(Generator source, float x, float y, float z)
```

`source` is the source generator.

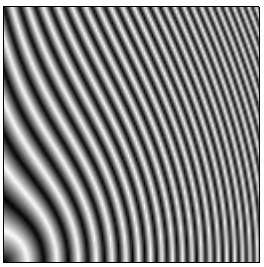
`v` is the scale vector, and `x,y,z` are its components. Negative scale will result in "mirroring" of source, and scale of 0 will turn it into a constant (effectively stretching infinitely).

Note that scaling is applied to *input coordinates,* not the noise itself. This means that high scale numbers increase frequency, and make the resulting pattern "smaller".

Perturb



Source



Result

This generator translates source noise by variable amount, depending on input coordinates. Translation vector in each point is determined by a user-supplied function. The pattern in the example picture was perturbed using the following

function:

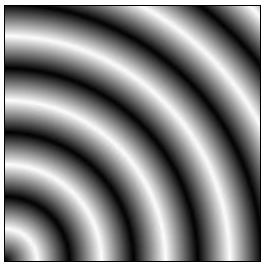
```
v=>new Vector3(
  v.x*v.x,
  v.y*v.x,
  v.z*v.x)
```

`Perturb` has one constructor:

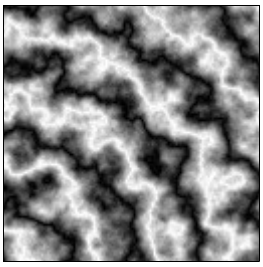
```
public Perturb(Generator source, Func<Vector3, Vector3> displacementSource)
```

`source` is the source generator, and `displacementSource` is the function that maps coordinates to displacements.

`Turbulence`



Source



Result

`Turbulence` is a special case of `Perturb` generator. It adds a random displacement to its source noise, using three different `PinkNoise` generators as a source of displacement values.

`Turbulence` has one constructor:

```
public Turbulence(Generator source, int seed)
```

`source` is the source generator, and `seed` is used to initialize displacing generators.

`Turbulence` defines three properties:

```
public float Power { get; set; }
```

```
public float Frequency { get; set; }
```

```
public int OctaveCount { get; set; }
```

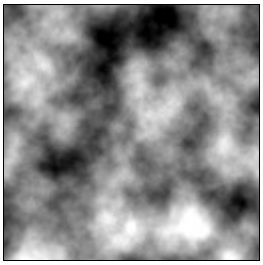
`Power` controls the amount of turbulence, i.e. how far the points are displaced. Default value is 1.

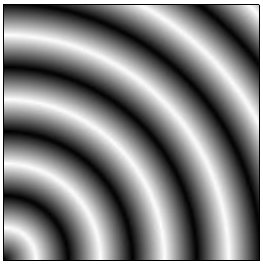
`Frequency` is the frequency of displacement generators, and thus the resulting "turbulent" pattern. Default value is 1.

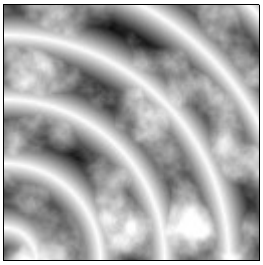
`OctaveCount` is the number of octaves in displacement generators. Default value is 6.

Combination

Max


Source (a)


Source (b)


Result

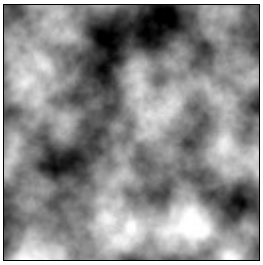
`Max` generator takes two source generators and returns maximum of two values.

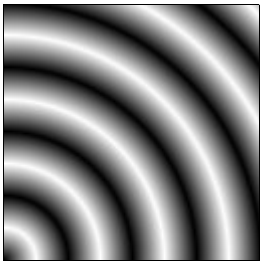
It has one constructor:

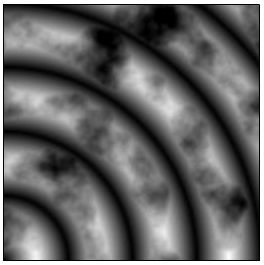
```
public Max(Generator a, Generator b)
```

`a` and `b` are source generators.

Min


Source (a)


Source (b)


Result

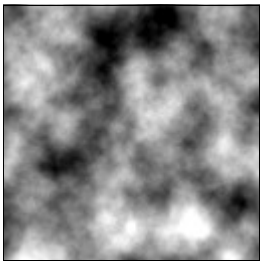
`Min` generator takes two source generators and returns minimum of two values.

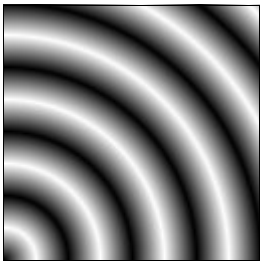
It has one constructor:

```
public Min(Generator a, Generator b)
```

`a` and `b` are source generators.

Add


Source (a)


Source (b)


Result (halved)

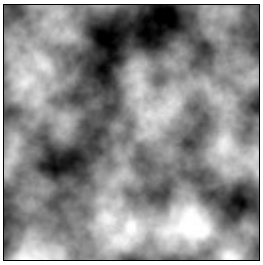
`Add` generator takes two source generators and returns sum of their values.

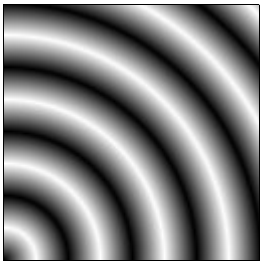
It has one constructor:

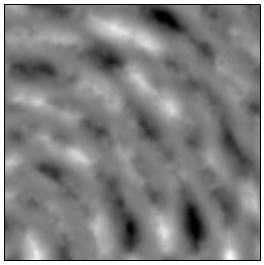
```
public Add(Generator a, Generator b)
```

`a` and `b` are source generators.

Multiply


Source (a)


Source (b)


Result

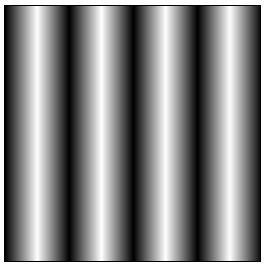
`Multiply` generator takes two source generators and returns product of their values.

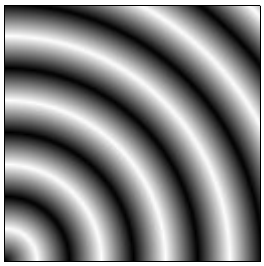
It has one constructor:

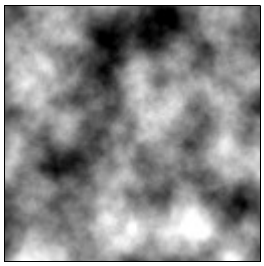
```
public Multiply(Generator a, Generator b)
```

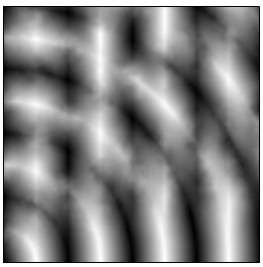
`a` and `b` are source generators.

Blend


Source (a)


Source (b)


Weight


Result

`Blend` generator takes three generators as input. Two of them are called *source* generators, and the third is called *weight*. Result of `Blend` is a value smoothly blended between first and second source values, as determined by weight. Where weight is 0 (or less) result is equal to first source value; where weight is 1 (or more) result is equal to second source value.

Note that weight values are clamped to [0,1] range, and most generators in CoherentNoise return values from -1 to 1. To change this range, you can divide the generator by 2 and add 0.5. `ScaleShift` method is a quick way to do just this.

`Blend` has one constructor:

```
public Blend(Generator a, Generator b, Generator weight)
```

`a` and `b` are source generators, and `weight` is weight generator.

Interpolation and S-curves

Interpolation is used in gradient and value noise generators, that often form the basis of most used noises. In CoherentNoise library, interpolation is defined by an `SCurve` class, that represents an S-shaped curve. The class has one abstract method:

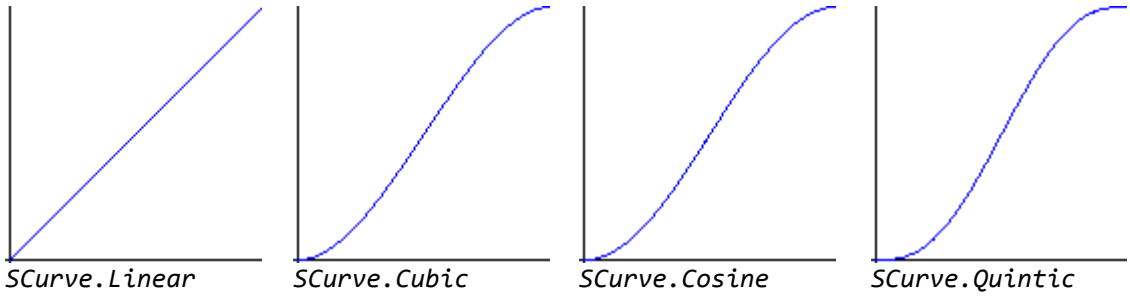
```
public abstract float Interpolate(float t);
```

This method takes a value between 0 and 1 and maps it to another value in the same range. Interpolation curves must map 0 to 0 and 1 to 1, and preferably also map values as smooth as possible (i.e. derivatives at 0 and 1 must be zero)

There are four classes derived from `SCurve`, defining four different algorithms. These can be accessed using static properties of `SCurve` class:

```
public static readonly SCurve Linear;

public static readonly SCurve Cubic;

public static readonly SCurve Quintic;

public static readonly SCurve Cosine;
```

The shapes of these curves are shown in the following picture:



Linear interpolation is the fastest, but it is prone to producing grid artifacts. Cubic is slower, but smoother; Cosine and Quintic are even better, but at the cost of more performance. Generally, cubic interpolation is a good compromise between quality and speed.

`SCurve` class also defines static property

```
public static SCurve Default { get; set; }
```

This property defines the interpolation algorithm to use by default (i.e. when it's not supplied in the noise constructor). It is initialized to `SCurve.Cubic`, but may be changed in case you want all your generators use some other method.

Texture creation

The TextureMaker class contains several static methods, useful when creating textures from noise.

```
public static Texture Make(
   int width,
   int height,
   Func<float, float, Color> colorFunc,
   TextureFormat format=TextureFormat.RGB24)
```

This is a generic texture creation method, that uses a user-supplied function to map coordinates to color value. The `colorFunc` function must return color given coordinates, that both range from 0 to 1. That is, pixel coordinates are mapped to float coordinates, so that (0,0) becomes (0,0) and (width, height) becomes (1,1). The `format` parameter defines desired texture format.

```
public static Texture MonochromeTexture(int width, int height, Generator noise)
```

This method creates a monochrome texture from noise. Noise generator is sampled in the Z=0 plane; and its values are clamped to [-1,1] range: in effect, values less than -1 become black, and more than 1 become white.

```
public static Texture AlphaTexture(int width, int height, Generator noise)
```

This method creates a alpha-only texture from noise. Noise generator is sampled in the Z=0 plane; and its values are clamped to [-1,1] range: in effect, values less than -1 become transparent, and more than 1 become opaque.

```
public static Texture RampTexture(int width, int height, Generator noise, Texture2D ramp)
```

This method creates a color texture using a *ramp* to map noise to color. A ramp is a one-dimensional texture that is sampled using noise value. As Unity3D has no 1D textures, this method actually samples the top line of ramp (i.e. points from (0,0) to (`ramp.width`, 0)). Noise value of -1 corresponds to (0,0) on ramp, and noise value of 1 – to (`ramp.width`, 0).

```
public static Texture BumpMap(int width, int height, Generator noise)
```

This method creates a bump map texture, using noise value as surface "height". Noise values are not clamped in this method, and in fact, increasing them (for example, multiplying noise by a constant) is a way to increase "bumpiness" of the resulting texture.

Chapter III

Example: clouds texture

As an example, let's generate a cloudy sky texture. Here's the code:

```
// Billow noise creates clouds
var billow = new BillowNoise(367567);
billow.Frequency = 4f;
billow.Persistence = 0.4f; // lower persistence: puffier clouds

var swirly = billow.Turbulence(0.8f, 0.05f); // turbulence to add some swirl

// Bias to make clouds bigger
var cloud = new Bias(swirly, 0.25f);

return TextureMaker.RampTexture(640, 640, cloud, Ramp);
```

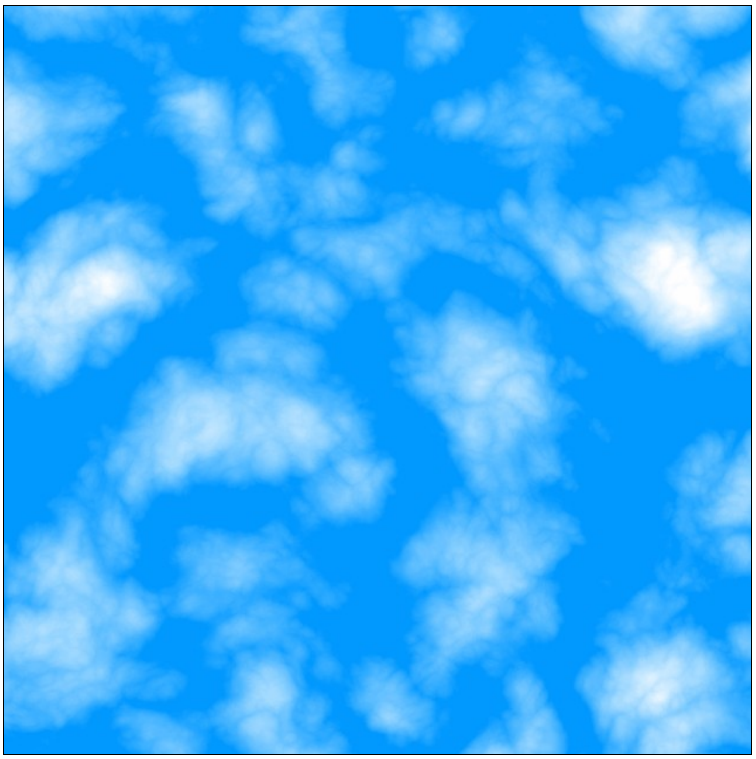
Code starts with a `BillowNoise`. It tends to generate isolated billowy shapes, with expanses of low values between them – exactly what we need for clouds. The seed for noise is just some number I made up. We then set noise parameters. `Frequency` determines the number of clouds: with higher frequencies, noise will be more "crammed", and more (smaller) shapes will fit into texture. `Persistence` affects "puffiness" of the clouds: with high persistence, cloud shapes will look too sharp, that's why we lower it a bit from the default value of 0.5.

Next, a little turbulence is added. Turbulence has the effect of making cloud pattern swirly – only a bit, since turbulence has low frequency and power.

Then we use `Bias`. `Bias` increases noise values, making cloud shapes bigger, but leaves lowest values still low – this ensures that we don't have a single "mega-cloud".

Finally, the last line simply turns our noise generator into a texture. Note that up to this point, no noise was generated: we simply created our generator, but never used it. The `TextureMaker` uses a `Ramp` texture to turn low noise values int blue colors (making "sky"), and high values into whites (making "clouds").

Here's the result:



Ramp texture used

Example: wood texture

Now let's make a more complex texture: wood. Here's code:

```
var circles = new Cylinders(1);

// grain noise
var grain = new PinkNoise(345623);
grain.Frequency = 16;
grain.OctaveCount = 4;

// stretch grain in Z direction
var scaledGrain = grain.Scale(1,1,0.25f);

// add grain to circles
var wood = 0.25f * scaledGrain + circles + 0.1f;

// perturb filtration:
var perturbedWood = wood.Turbulence(0.5f,0.1f);

// prepare a “slice”:
var slice = perturbedWood.
      Translate(-3f, 1.5f, 1.5f).
      Rotate(80, 20, 45).
      Scale(16, 16, 16);

var texture = TextureMaker.RampTexture(640, 640, slice, Ramp);
var bump = TextureMaker.BumpMap(640, 640, slice);
```

This example takes advantage of 3D functionality of `CoherentNoise` generators. We'll created a tree-like 3D pattern, and then “slice” a texture out of it. We start not with a noise, but with a cylindrical pattern. This will be growth rings pattern of our “tree”.

Next we create some `PinkNoise` that would simulate wood grain. It mainly serves to add some irregularity into the pattern. We scale this noise along Z axis (this is the axis parallel to our “tree”).

Grain and circular pattern are added together, with grain given lower weight (0.25) This way, the pattern stays mainly circular, but with irregularities.

0.1 is also added to increase noise values. This will make “low” rings thinner, and “high” rings wider.

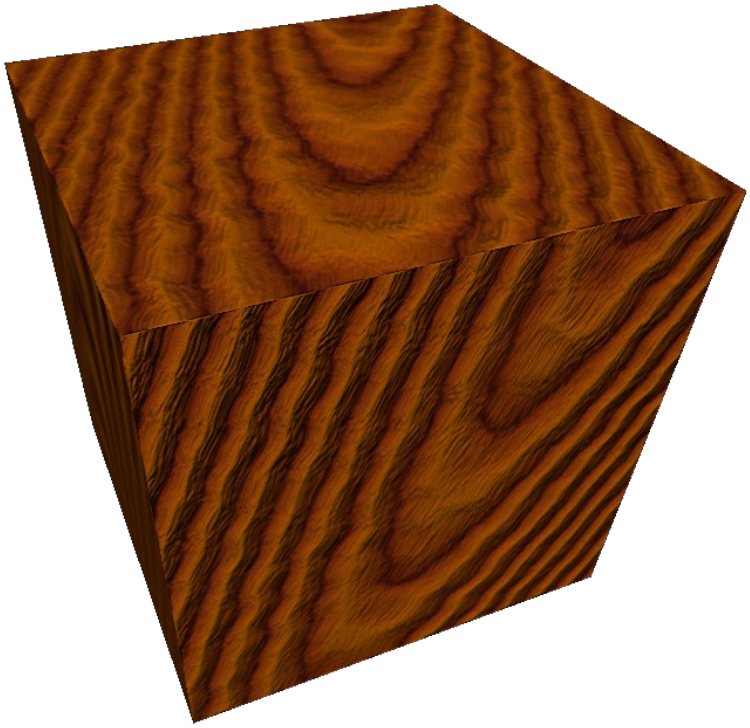
Next, we add some turbulence. Real growth rings are not perfect circles, and turbulence will bend our circles randomly to make them more natural.

In the end, we prepare our “slice”. `TextureMaker` always uses the [0,1]x[0,1] region in the Z=0 plane. To make our slice more interesting, we translate, rotate and scale noise pattern: this has the same effect as translating, rotating, and

scaling the sampling region, only in opposite direction.

Finally, we use `TextureMaker` to create not just one, but two textures: one actual texture and one bump-map. We can then link these two textures to a material that supports bump-mapping.

Here's what we've made:



Ramp texture used