Gradient Noise — Detailed Overview and C# Implementation Guide

A ready-to-use `.md` reference for a .NET C# project. Contains: conceptual overviews, formulas, input parameters, usage notes, and C# example signatures/implementations for several gradient-noise families.

Table of contents

---

1. Perlin Noise (Classic)

Overview

Perlin noise is a gradient noise algorithm invented by Ken Perlin (1983). It assigns pseudorandom gradient vectors at lattice points on an integer grid and interpolates values for arbitrary sample points by using the dot product between the gradient and the distance vector. The result is a smooth, band-limited-looking noise often used for textures and terrain.

Formula (2D)

1. Let `x,y` be the sample point.
2. Determine the integer lattice corners: `x0 = floor(x)`, `x1 = x0 + 1`, same for `y0`, `y1`.
3. Compute distance vectors to each corner: `dx = x - xi`, `dy = y - yi`.
4. For each corner `(xi, yi)` pick a gradient vector `g(xi, yi)` and compute the dot product `d = g dot (dx, dy)`.
5. Interpolate the four dot products using a smooth fade function `f(t)` (commonly `6t^5 - 15t^4 + 10t^3`).

In symbols, for corner contributions:

```
C00 = g(x0,y0) · (x - x0, y - y0)
C10 = g(x1,y0) · (x - x1, y - y0)
C01 = g(x0,y1) · (x - x0, y - y1)
C11 = g(x1,y1) · (x - x1, y - y1)

u = fade(x - x0)
v = fade(y - y0)

ix0 = lerp(C00, C10, u)
ix1 = lerp(C01, C11, u)
value = lerp(ix0, ix1, v)
```

`lerp(a,b,t) = a + t*(b-a)`

Inputs / Parameters

- `float x, y` — sample coordinates (can be fractional)
- `float frequency` — scales input coordinates (higher freq = finer detail)
- `int seed` — permutation/seed for pseudo-random gradients
- `int octaves` — number of layers for fractal noise
- `float persistence` — amplitude multiplier between octaves
- `float lacunarity` — frequency multiplier between octaves

Implementation notes for C

- Use a permutation table (size 256) repeated to avoid modulo branch costs.
- Store precomputed gradient vectors (e.g., 8 directions for 2D) as `Vector2[]`.
- Use the quintic fade `f(t) = 6t^5 - 15t^4 + 10t^3` for smooth derivatives.
- Return noise in `[-1, 1]` and optionally remap to `[0,1]`.

Simple C# signature example

```
// Single octave, 2D Perlin
public static float Perlin2D(float x, float y, int seed = 0, float frequency =
1f)
{
    // implementation
}

// Fractal (fBm)
public static float Perlin2DFractal(float x, float y, int seed = 0, int octaves
= 4, float lacunarity = 2f, float persistence = 0.5f)
{
    float amplitude = 1f; float frequencyLocal = frequency; float sum = 0f;
float max = 0f;
    for (int i = 0; i < octaves; i++) {
```

```
        sum += Perlin2D(x * frequencyLocal, y * frequencyLocal, seed + i) *
amplitude;
        max += amplitude;
        amplitude *= persistence; frequencyLocal *= lacunarity;
    }
    return sum / max; // normalized
}
```

---

2. Improved Perlin Noise (2002)

Overview

Improvements on classic Perlin noise that reduce directional artifacts and simplify the gradient selection/permutation logic. Uses a carefully chosen gradient set and a fixed permutation array.

Differences vs Classic

- Uses a better permutation table and a specific set of gradient vectors (12 in 3D, 8 in 2D) to reduce artifacts.
- Implementation details (hashing & gradients) borrowed from Perlin's 2002 reference implementation.

Formula & Notes

Same math flow as classic Perlin (dot products + quintic fade + bilinear/trilinear interpolation), but with an improved gradient selection using a consistent permutation table.

Inputs / Parameters

Same as Perlin. Seed often implemented by shuffling or rotating the permutation table.

C# implementation notes

- You can seed by permuting the base permutation array using a seeded RNG.
- Keep gradient arrays typed as `float[]` or `Vector2[]` for performance.

---

3. Simplex Noise

Overview

Simplex noise (Perlin, 2001) operates on simplices (triangles in 2D, tetrahedra in 3D). It has fewer grid-aligned artifacts and is more efficient at higher dimensions. It computes contributions only from simplex corners; the number of corners scales linearly with dimension (vs. 2^n for grid methods).

High-level formula (2D)

1. Skew input space to place it on a simplex grid: `s = (x + y) * F2` where `F2 = 0.5 * (sqrt(3.0) - 1.0)`.
2. Determine simplex cell indices and unskew the cell origin using `G2 = (3.0 - sqrt(3.0)) / 6.0`.
3. For each corner of the simplex (3 corners in 2D) compute the corner offset, select a gradient, compute `t = 0.5 - x^2 - y^2`; if `t > 0` compute contribution `t^4 * (g dot (x,y))`.
4. Sum corner contributions and scale to desired range.

Exact constants and skew/unskew math are required for correct shape.

Inputs / Parameters

Same as Perlin (x, y, frequency, seed, octaves, etc.).

Implementation notes for C

- Simplex is more complex to implement but returns better results in higher dimensions.
- Use integer hashing via permutation table to choose gradients.
- Keep constants `F2`, `G2`, `F3`, `G3` as `const float`.

C# signature example

```
public static float Simplex2D(float x, float y, int seed = 0, float frequency =
1f)
{
    // implementation uses skew/unskew constants
}
```

---

4. OpenSimplex Noise

Overview

OpenSimplex is an alternative to Simplex designed to avoid earlier patent issues and produce visually pleasing noise. It uses a different lattice and gradient generation strategy which yields a slightly different visual character.

Inputs / Parameters

Same parameters. Implementation details differ (kernel shapes, gradients) and typically require a full port of the OpenSimplex algorithm.

C# notes

- Several C# ports exist; for production code either port a well-reviewed implementation or translate the algorithm carefully.
- Performance and visual differences are subtle—test both Simplex and OpenSimplex to decide.

---

5. Gradient Value Noise (Hybrid)

Overview

Value noise assigns a pseudo-random scalar at lattice points and interpolates values. Gradient value noise blends ideas: use gradients but interpolate like value noise or vice versa. This can be used to control sharpness or band-limiting.

Formula (concept)

- Value noise: `v00 = valueAt(x0,y0)` etc., then `value = bilinearInterp(v00, v10, v01, v11, u, v)`
- Gradient-value hybrid: compute gradient contributions but optionally blend them with value-based interpolation to get different spectral characteristics.

Inputs / Parameters

- `float x, y`, `seed`, `frequency`
- `interpolation` type: linear, cubic, quintic (affects smoothness)

C# ideas

- Reuse permutation table and random-value table.
- Expose `enum Interpolation { Linear, Smoothstep, Quintic }` to let caller choose interpolation function.

---

6. Gabor Noise (Overview)

Overview

Gabor noise is generated by placing a random set of Gabor kernels (sinusoid modulated by a Gaussian) at random positions and summing their responses. It's computationally heavier but excellent for generating certain procedural textures and for controlling spectral properties precisely.

Formula (high level)

A single 2D Gabor kernel at position `x0` with orientation `w` and frequency `k`:

```
G(x) = exp(-|x-x0|^2 / (2 * sigma^2)) * cos(k · (x - x0) + phase)
```

Sum many such kernels placed by a Poisson process.

Inputs / Parameters

- `density` (# kernels per area), `sigma` (gaussian width), `k` (frequency vector), `seed`

C# notes

- Expensive to sum many kernels at runtime; often precompute or use a tileable texture.
- Good for specialized textures (wood, fabric, etc.).

---

7. Cellular / Worley Noise + Gradient Blends

Overview

Worley (cellular) noise is not gradient noise but pairs well with gradient interpolation. It computes distance to nearest feature points in a cell and yields patterns of cells. Blending Worley outputs with gradient-based noise lets you get cracked-earth, pebbles, or organic cell structures with smooth transitions.

Key functions

- Distance metrics: Euclidean, Manhattan, Chebyshev.
- `F1`, `F2`, `F3` — distances to 1st, 2nd, 3rd nearest feature points.

Inputs / Parameters

- `float x,y`, `density`, `seed`, `distanceFunction`, `jitter`

C# notes

- Use integer cell loops around the sample cell (e.g., search -1..+1 neighbors) to find nearest features.
- Combine with `Perlin2D` or `Simplex2D` by e.g. mixing `lerp(perlin, worley, mask)`.

---

8. Utility Functions & Parameter Glossary

- `fade(t)` — smoothing function for interpolation; commonly `6t^5 - 15t^4 + 10t^3`.
- `lerp(a,b,t)` — linear interpolation, `a + t*(b-a)`.
- `hash(i, j, seed)` — deterministic integer hash for lattice coordinates (use permutation table or fast integer hash).
- `grad(hash, dx, dy)` — select a gradient vector based on hash and compute dot product.
- `remapTo01(v)` — `(v + 1) * 0.5f` to convert `[-1,1]` to `[0,1]`.

Parameter meanings: - `frequency` — spatial scaling; multiply coordinates by frequency for more detail. - `octaves` — layers in fractal sum (fBm). - `persistence` — amplitude reduction each octave. - `lacunarity` — frequency increase each octave. - `seed` — deterministic randomness.

---

9. Example C# API Design

A small, clean class design you can copy into your .NET project:

```
public static class Noise2D
{
    // Public API
    public static float Perlin(float x, float y, int seed = 0, float frequency
= 1f);
    public static float PerlinFractal(float x, float y, int seed = 0, int
octaves = 4, float lacunarity = 2f, float persistence = 0.5f);

    public static float Simplex(float x, float y, int seed = 0, float frequency
= 1f);
    public static float OpenSimplex(float x, float y, int seed = 0, float
frequency = 1f);

    public static float ValueNoise(float x, float y, int seed = 0,
Interpolation interp = Interpolation.Quintic);

    public static float Worley(float x, float y, int seed = 0, float density =
1f, DistanceMetric metric = DistanceMetric.Euclidean);

    // Utility enums
    public enum Interpolation { Linear, Smoothstep, Quintic }
    public enum DistanceMetric { Euclidean, Manhattan, Chebyshev }
}
```

Include internal/private methods for hashing, permutation table generation, and gradient arrays. Keep all arrays `static readonly` and thread-safe (immutable) to be safe in multithreaded apps.

---

10. Performance Tips & Testing

- Use `float` instead of `double` for speed on GPUs and many CPU workloads.
- Avoid allocations in inner loops; reuse arrays or buffers.
- Precompute permutation table and gradient vectors.
- For fractal noise, precompute frequency and amplitude multipliers for each octave.
- Use SIMD (System.Numerics.Vector2/Vector<float>) if you need to evaluate many samples.
- For shaders, port a simplified implementation (GLSL/HLSL) and compare outputs.

Appendix: Small Code Samples

Fade function

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static float Fade(float t) => t * t * t * (t * (t * 6 - 15) + 10);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static float Lerp(float a, float b, float t) => a + t * (b - a);
```

Simple hashing (permutation style)

```
private static readonly byte[] Perm = BuildPermutationTable(seedSomehow);
private static int Hash(int x, int y) => Perm[(Perm[x & 255] + y) & 255];
```

Next steps / Suggested files to add to your repo

- `Noise2D.cs` — public API + Perlin implementation
- `SimplexNoise.cs` — Simplex implementation
- `OpenSimplex.cs` — optional, if you need an alternative
- `WorleyNoise.cs` — cellular noise
- `NoiseTests.cs` — Unit tests that assert value ranges, tiling, and visual output samples

---

If you want, I can now: - Provide full, copy-pasteable C# implementations for **Perlin** and **Simplex** (single file each). - Create unit tests and example images (heatmaps) using `System.Drawing`.

Tell me which implementation(s) you want next and I will add the C# source files.