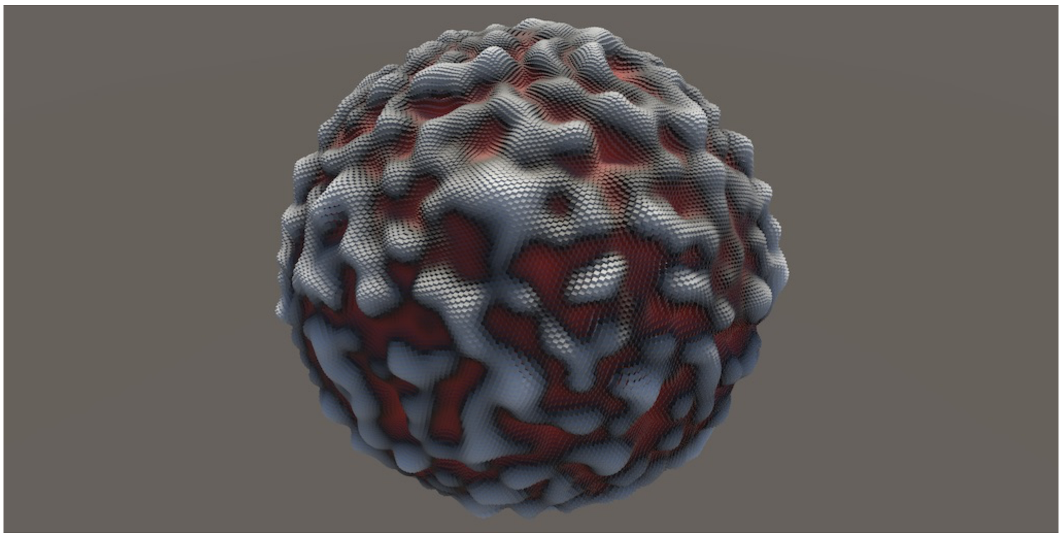
published 2021-06-18

Perlin Noise Gradient Noise

Make lattice noise generic.
Add support for gradient noise.
Generate 1D, 2D, and 3D Perlin noise.

This is the fourth tutorial in a series about pseudorandom noise. It enhances our lattice noise job to also support Perlin noise.

This tutorial is made with Unity 2020.3.6f1.



A sphere showing 3D Perlin noise.

1 Generic Gradient Noise

Value noise is lattice noise that defines constant values at the lattice points. Interpolation of these values produces a smooth pattern, but the lattice is still quite obvious. An alternative approach is to interpolate between functions instead of constant values. This means that each lattice point has its own function. To keep it simple and uniform all points should get the same kind of function, only the parameterization varies. The simplest function is a constant value, which would be Value noise. One step up from that would be a function that is linearly dependent on the sample coordinates relative to the lattice point. The most straightforward is $f(x) = x$ where x is the relative coordinate. This would produce linear ramps or gradients centered on lattice points. Hence this type of noise is known as gradient noise.

As Value noise can be considered a trivial case of gradient noise let's adjust our lattice noise so it always functions as gradient noise.

1.1 Gradient Interface

We begin by introducing a new partial class asset for `Noise`, this time named *Noise.Gradient*. In it we declare the `IGradient` interface.

```
using Unity.Mathematics;

using static Unity.Mathematics.math;

public static partial class Noise {

	public interface IGradient {}
}
```

The purpose of a gradient is to evaluate a function, with a hash and a relative coordinate as parameters. Define a vectorized `Evaluate` method signature for this, initially for 1D noise, which means that besides the hash it needs to have parameter for the X coordinates.

```
	public interface IGradient {
		float4 Evaluate (SmallXXHash4 hash, float4 x);
	}
```

Now we can add a `Value` struct type in the same partial class that implements `IGradient` for 1D Value noise. It simply ignores the coordinates and returns the A floats of the hash, like our lattice noise currently does.

```
	public struct Value : IGradient {

		public float4 Evaluate (SmallXXHash4 hash, float4 x) => hash.Floats01A;
	}
```

To also support 2D and 3D noise add `Evaluate` variants with two and three coordinate parameters to `IGradient`.

```
	public interface IGradient {
		float4 Evaluate (SmallXXHash4 hash, float4 x);

		float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y);

		float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y, float4 z);
	}
```

`Value` must implement these methods as well, again by simply returning the hash value and ignoring the coordinates. The unused parameters will not affect performance because *Burst* eliminates all method invocation overhead and unused values.

```
	public struct Value : IGradient {

		public float4 Evaluate (SmallXXHash4 hash, float4 x) => hash.Floats01A;

		public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y) => hash.Floats01A;

		public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y, float4 z) =>
			hash.Floats01A;
	}
```

1.2 Gradient Input

To provide the relative coordinates for gradients we'll add a vectorized float field named `g` to `LatticeSpan4` in *Noise.Lattice*.

```
	struct LatticeSpan4 {
		public int4 p0, p1;
		public float4 g;
		public float4 t;
	}
```

We find the relative coordinates by subtracting `p0` from the coordinates in `GetLatticeSpan4`.

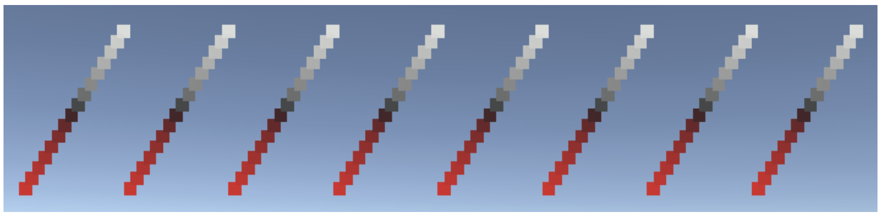
```
		span.p0 = (int4)points;
		span.p1 = span.p0 + 1;
		span.g = coordinates - span.p0;
```

Now we can adjust `Lattice1D.GetNoise4` so it invokes `Evaluate` on `Value` instead of directly getting a value from the hash. Pass the relevant hash and the relative coordinates as arguments to `Evaluate`.

```
	public struct Lattice1D : INoise {

		public float4 GetNoise4(float4x3 positions, SmallXXHash4 hash) {
			LatticeSpan4 x = GetLatticeSpan4(positions.c0);

			var g = default(Value);
			return lerp(
				g.Evaluate(hash.Eat(x.p0), x.g), g.Evaluate(hash.Eat(x.p1), x.g), x.t
			) * 2f - 1f;
		}
	}
```

At this point we still get the same result as before for 1D Value noise and the generated assembly code for the job is also exactly the same. But to get an idea of how gradient noise works we can temporarily change `Value.Evaluate` so it becomes the simple gradient function $f(x) = x$.

```
		public float4 Evaluate (SmallXXHash4 hash, float4 x) => x;
```



Raw gradient; 1D noise on a plane; resolution 128.

The resulting pattern is a sequence of linear 1D ramps that go from -1 to 1 in between lattice points. There is no apparent blending because we use the exact same gradient on both sides of each span, relative to `p0` on both ends. To turn it into proper gradient noise the second gradient must be based on coordinates relative to `p1`. So replace the single `g` field in `LatticeSpan4` with `g0` and add a new `g1` field.

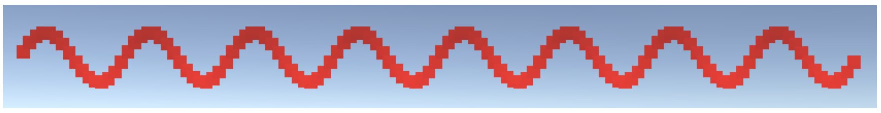
```
	struct LatticeSpan4 {
		public int4 p0, p1;
		public float4 g0, g1;
		public float4 t;
	}
```

Because the second lattice point of a span sits one unit further along that dimension, we find `g1` in `GetLatticeSpan4` by subtracting one from `g0`.

```
		span.p0 = (int4)points;
		span.p1 = span.p0 + 1;
		span.g0 = coordinates - span.p0;
		span.g1 = span.g0 - 1f;
```

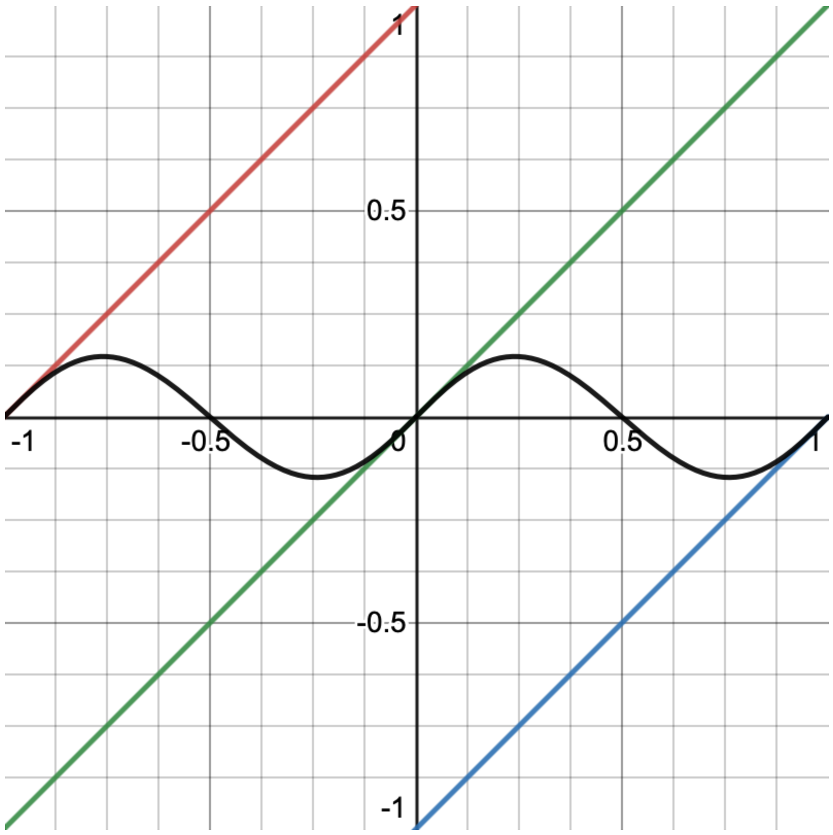
Adjust `Lattice1D.GetNoise4` so it uses the correct relative coordinates: `p0` together with `g0` and `p1` together with `g1`.

```
			return lerp(
				g.Evaluate(hash.Eat(x.p0), x.g0), g.Evaluate(hash.Eat(x.p1), x.g1), x.t
			) * 2f - 1f;
```



Interpolated gradients.

If we had used linear interpolation the resulting pattern would be flat because the gradients would have canceled each other. But because we use smooth C2-continuous interpolation each gradient is dominant near its lattice point and we get a wave pattern.



Interpolating three gradients across two spans.

Note that the gradients are zero at their lattice points and that the interpolated pattern is also zero in the middle of a span, where the gradients cancel each other.

The pattern that we get is wholly negative because we apply a range adjustment afterwards. This adjustment is specific to Value noise, so let's restore the noise and move the adjustment to the `Value` methods.

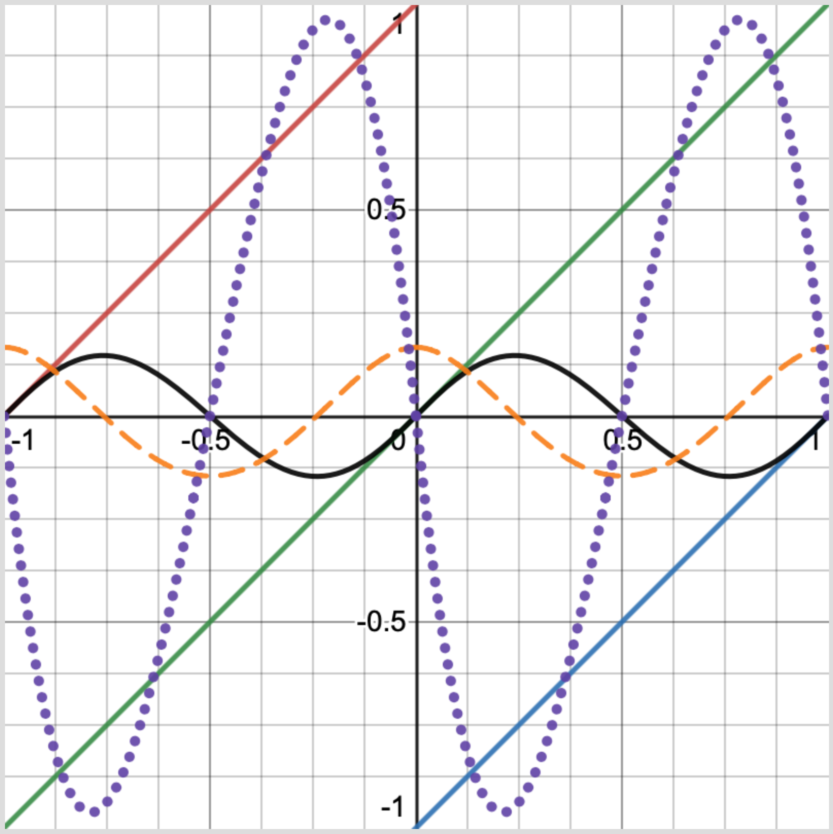
```
	public struct Value : IGradient {

		public float4 Evaluate (SmallXXHash4 hash, float4 x) => hash.Floats01A * 2f - 1f;

		public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y) =>
			hash.Floats01A * 2f - 1f;

		public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y, float4 z) =>
			hash.Floats01A * 2f - 1f;
	}
```

Then remove it from `Lattice1D.GetNoise4`.

```
			return lerp(
				g.Evaluate(hash.Eat(x.p0), x.g0), g.Evaluate(hash.Eat(x.p1), x.g1), x.t
			);
			~~//) * 2f - 1f;~~
```

Is the interpolation of gradients still C2-continuous?

Because gradients have a slope at the lattice points the first derivative is no longer zero there, but it is still continuous across spans because the gradient is the same on both sides. The second derivative is indeed still zero at the lattice points and thus also continuous.



Interpolated gradients with 1st and 2nd derivatives, divided by 6 to fit.

1.3 Generic Lattice

Now that we have a gradient version of Value noise we can make `Lattice1D` generic by giving it an `IGradient` struct type parameter, like we did for `Noise.Job`.

```
    public struct Lattice1D<G> : INoise where G : struct, IGradient {

        public float4 GetNoise4(float4x3 positions, SmallXXHash4 hash) {
            LatticeSpan4 x = GetLatticeSpan4(positions.c0);

            var g = default(G);
            return lerp(
                g.Evaluate(hash.Eat(x.p0), x.g0), g.Evaluate(hash.Eat(x.p1), x.g1), x.t
            );
        }
    }
```

Adjustment `Lattice2D` in the same way, this time using the 2D variant of `IGradient.Evaluate`. Make sure to use the correct relative coordinates for all lattice points.

```
	public struct Lattice2D<G> : INoise where G: struct, IGradient {

		public float4 GetNoise4 (float4x3 positions, SmallXXHash4 hash) {
			…

			var g = default(G);
			return lerp(
				lerp(
					g.Evaluate(h0.Eat(z.p0), x.g0, z.g0),
					g.Evaluate(h0.Eat(z.p1), x.g0, z.g1),
					z.t
				),
				lerp(
					g.Evaluate(h1.Eat(z.p0), x.g1, z.g0),
					g.Evaluate(h1.Eat(z.p1), x.g1, z.g1),
					z.t
				),
				x.t
			);
			~~//) * 2f - 1f;~~
		}
	}
```

And update `Lattice3D` as well.

```
	public struct Lattice3D<G> : INoise where G : struct, IGradient {

		public float4 GetNoise4 (float4x3 positions, SmallXXHash4 hash) {
			…

			var g = default(G);
			return lerp(
				lerp(
					lerp(
						g.Evaluate(h00.Eat(z.p0), x.g0, y.g0, z.g0),
						g.Evaluate(h00.Eat(z.p1), x.g0, y.g0, z.g1),
						z.t
					),
					lerp(
						g.Evaluate(h01.Eat(z.p0), x.g0, y.g1, z.g0),
						g.Evaluate(h01.Eat(z.p1), x.g0, y.g1, z.g1),
						z.t
					),
					y.t
				),
				lerp(
					lerp(
						g.Evaluate(h10.Eat(z.p0), x.g1, y.g0, z.g0),
						g.Evaluate(h10.Eat(z.p1), x.g1, y.g0, z.g1),
						z.t
					),
					lerp(
						g.Evaluate(h11.Eat(z.p0), x.g1, y.g1, z.g0),
						g.Evaluate(h11.Eat(z.p1), x.g1, y.g1, z.g1),
						z.t
					),
					y.t
				),
				x.t
			);
			~~//) * 2f - 1f;~~
		}
	}
```

We now have to explicitly declare that we are using the Value noise versions of lattice noise jobs in `NoiseVisualization`.

```
	static ScheduleDelegate[] noiseJobs = {
		Job<Lattice1D<Value>>.ScheduleParallel,
		Job<Lattice2D<Value>>.ScheduleParallel,
		Job<Lattice3D<Value>>.ScheduleParallel
	};
```

Everything works the same as before, but it is now possible to add other gradient noise types with only a little bit of extra code.

2 Perlin Noise

Ken Perlin came up with the first version of gradient noise, hence this classical version of noise is known as Perlin noise. Compared to what we already know of gradient noise at this point, Perlin noise adds the idea that gradient vectors can have different orientations. The interpolation of these different gradients produces a more varied and less blocky pattern than Value noise.

Isn't Perlin noise based on a permutation table?

Yes, but a permutation table is just one way of generating pseudorandom values for lattice points. It works well for simple hardware but suffers from a small domain and cannot be seeded. It also doesn't vectorize because it requires multiple array lookups, for which there are no SIMD instructions. So we base it on `SmallXXHash4` instead.

2.1 Second Gradient Noise Type

To implement Perlin noise, add a `Perlin` struct type to *Noise.Gradient* that implements `IGradient`, just like `Value`. Intially set all evaluations to zero.

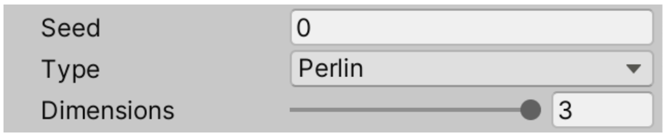
```
	public struct Perlin : IGradient {

		public float4 Evaluate (SmallXXHash4 hash, float4 x) => 0f;

		public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y) => 0f;

		public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y, float4 z) => 0f;
	}
```

Then turn the noise jobs array in `NoiseVisualization` into a two-dimensional array. Elements of such arrays have two index components, so the array type changes from `ScheduleDelegate[]` to `ScheduleDelegate[,]`. Then wrap the existing initialization in a second set of curly braces and insert a new set for the Perlin noise jobs before the Value noise set.

```
	static ScheduleDelegate[,] noiseJobs = {
		{
			Job<Lattice1D<Perlin>>.ScheduleParallel,
			Job<Lattice2D<Perlin>>.ScheduleParallel,
			Job<Lattice3D<Perlin>>.ScheduleParallel
		},
		{
			Job<Lattice1D<Value>>.ScheduleParallel,
			Job<Lattice2D<Value>>.ScheduleParallel,
			Job<Lattice3D<Value>>.ScheduleParallel
		}
	};
```

To make switching between the noise types possible add an enum configuration field for Perlin and Value noise and use it as the first array index argument in `UpdateVisualization`.

```
	public enum NoiseType { Perlin, Value }

	[SerializeField]
	NoiseType type;

	…

	protected override void UpdateVisualization (
		NativeArray<float3x4> positions, int resolution, JobHandle handle
	) {
		noiseJobs[(int)type, dimensions - 1](
			positions, noise, seed, domain, resolution, handle
		).Complete();
		noiseBuffer.SetData(noise);
	}
```


Seed 0
Type Perlin
Dimensions 3

Noise type configuration option.

2.2 1D Gradients

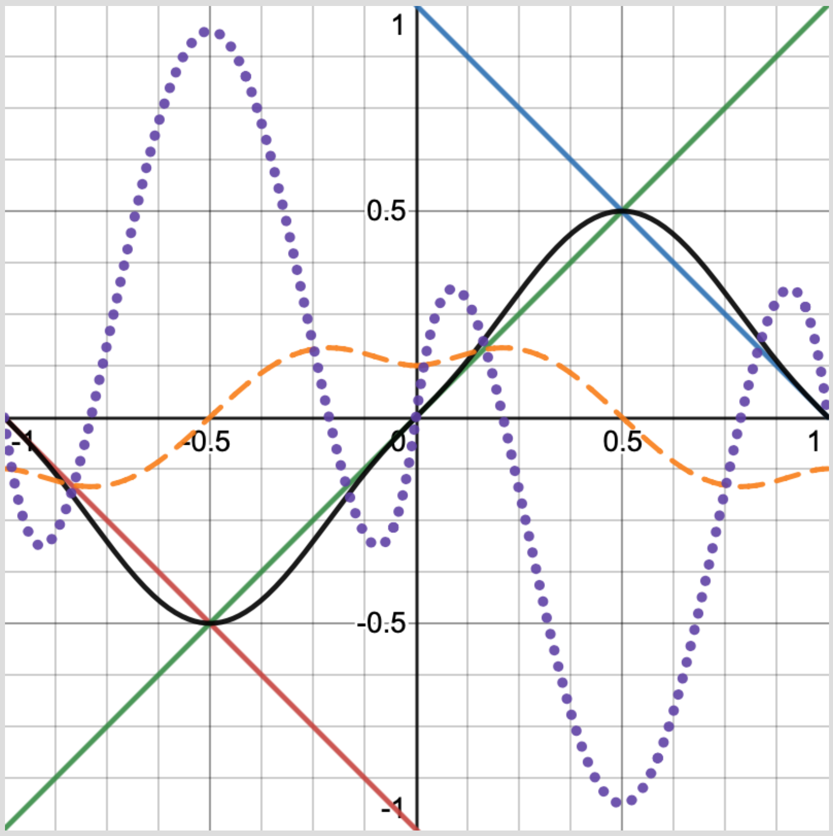
Ken Perlin never made a proper 1D noise variant, because it isn't very useful, but we do because it makes understanding higher dimensions easier. We already tested 1D gradient noise earlier, with the fixed gradient function $f(x) = x$. The idea of Perlin noise is that the gradients of lattice points can be different. In the case of 1D the most obvious other gradient would simple be the negative version of what we already used: $f(x) = -x$. We can use the first bit of the hash to determine whether we select the positive or negative version.

```
	public struct Perlin : IGradient {

		public float4 Evaluate (SmallXXHash4 hash, float4 x) =>
			select(-x, x, ((uint4)hash & 1) == 0);

		…
	}
```

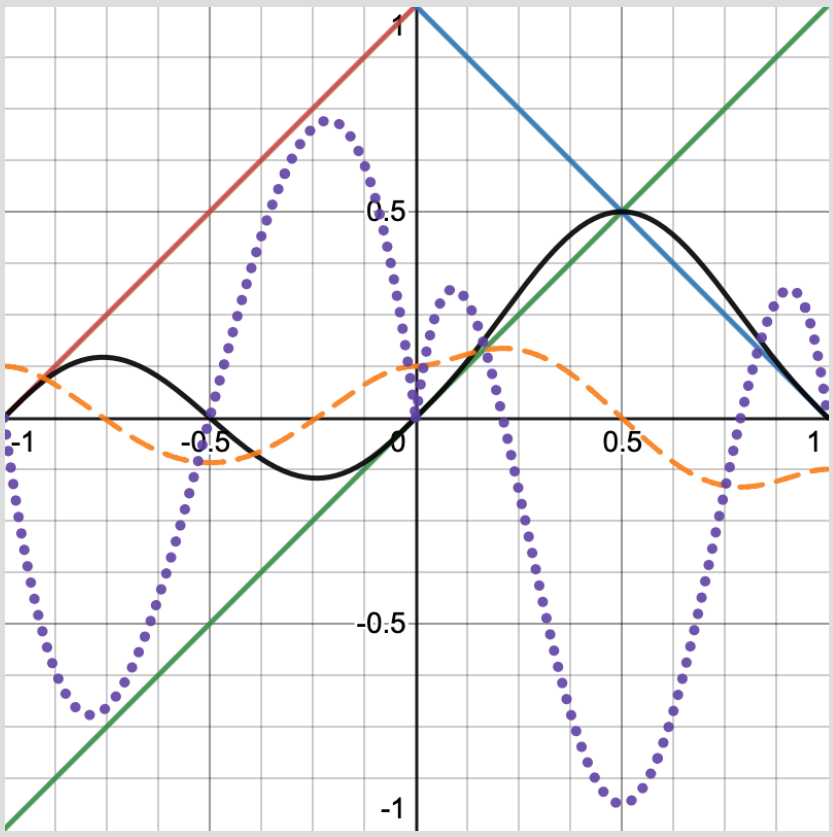
This leads to four possible gradient interpolations per lattice span: positive-positive, negative-negative, positive-negative, and negative-negative. As positive and negative mirror each other there are only two unique cases: same and different gradients.

Is the noise still C2-continuous when using varying gradients?

Yes. The second derivative is always zero at the lattice points and the first derivative is continuous because the same gradient is still used on both sides of the lattice point.



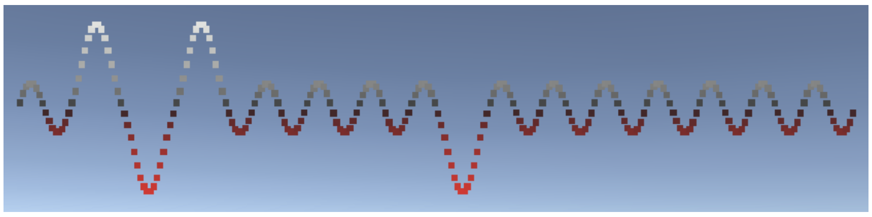
Negative-positive-negative gradients, derivatives divided by 8 to fit.



Positive-positive-negative gradients, derivatives divided by 8 to fit.

The noise reaches a maximum amplitude of 0.5 halfway between lattice points with opposite gradients. This is the case because both are 0.5 at that point and we end up with their average. Ideally the noise has a maximum amplitude of 1, which we can achieve by doubling the gradients.

```
		public float4 Evaluate (SmallXXHash4 hash, float4 x) =>
			2f * select(-x, x, ((uint4)hash & 1) == 0);
```

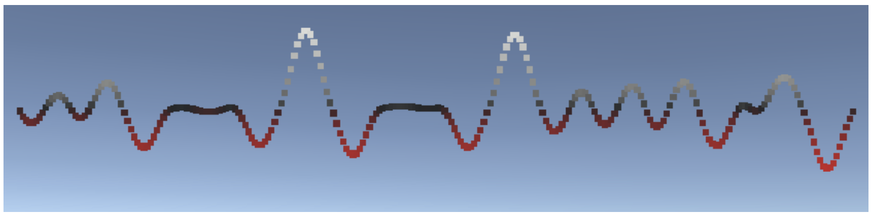

1D binary Perlin noise; domains scale 16; resolution 256.

2.3 Variable Gradients

I name the noise that we get at this point binary Perlin noise, because its gradients can only have two states. Thus the noise consists of sequences of gradients that all point in the same direction, except when there is a sign flip. The flips show up as maximum-amplitude waves while the rest of the pattern consists of small identical small waves. This looks very rigid, so let's try a different approach.

Instead of performing a binary selection, use the hash to scale the relative coordinates by a factor in the −1–1 range. We still need to double that to achieve a maximum amplitude of 1.

```
		public float4 Evaluate (SmallXXHash4 hash, float4 x) =>
			2f * (hash.Floats01A * 2f - 1f) * x;
```


Variable gradients.

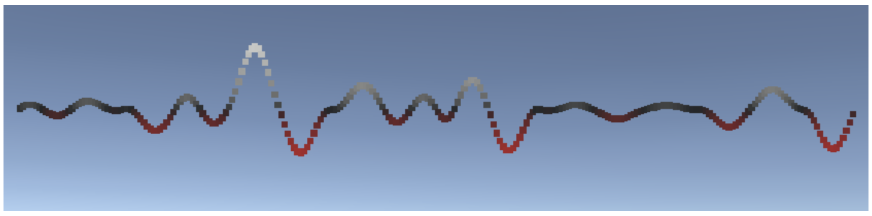
This looks more interesting, as we get a large variety of gradients with different amplitudes, both positive and negative. This does make it far less likely that the maximum amplitude is reached, so the average amplitude of the noise is reduced.

We can also combine the variable and the binary approach, using the first bit to select the sign and scaling by float A.

```
	public float4 Evaluate (SmallXXHash4 hash, float4 x) =>
		2f * hash.Floats01A * select(-x, x, ((uint4)hash & 1) == 0);
```

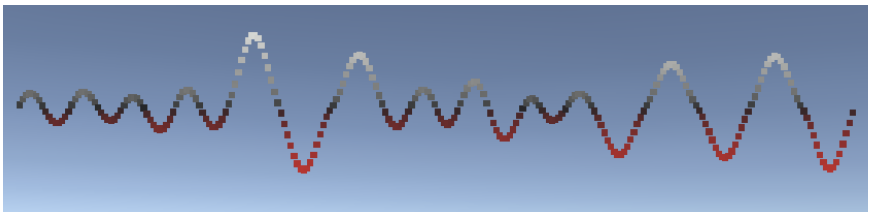
But that would include the first bit in both choices, which introduces a dependency. To keep the choices independent use the ninth bit to determine the sign instead.

```
		2f * hash.Floats01A * select(-x, x, ((uint4)hash & 1 << 8) == 0);
```


Different variable gradients.

The advantage of this approach is that we can introduce a minimum amplitude, regardless of the gradient direction. This way we can prevent degenerate areas from appearing, where the scale ends up close to zero for multiple successive lattice points, which would produce a flat region. The simplest approach is to set the minimum amplitude to 1 and add the hash float to that. The result can be considered a mix of the binary and variable approaches, guaranteeing a minimum wave amplitude but adding some variety on top of that.

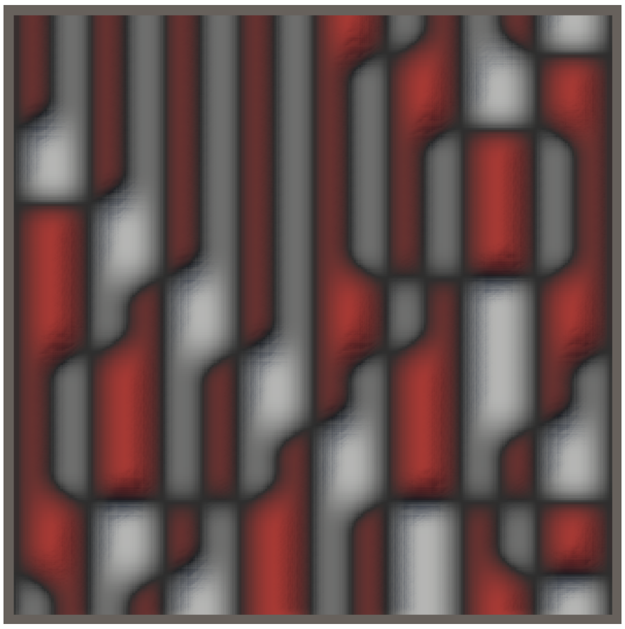
```
	public float4 Evaluate (SmallXXHash4 hash, float4 x) =>
		(1f + hash.Floats01A) * select(-x, x, ((uint4)hash & 1 << 8) == 0);
```


Binary–variable mix.

2.4 2D Gradients

Moving on to 2D Perlin noise, we again start with binary gradients in a single dimension, based on the first hash bit, and see how that looks.

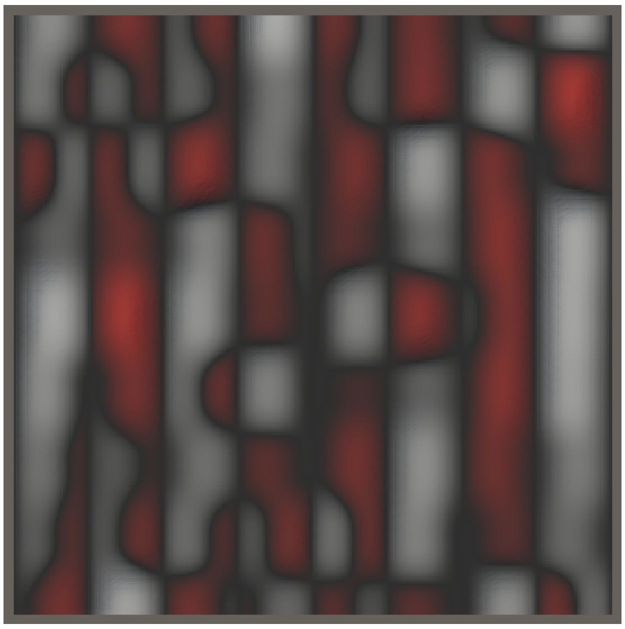
```
	public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y) =>
		select(-x, x, ((uint4)hash & 1) == 0);
```



2D binary Perlin noise plane; domain scale 8; top-down view.

The result is an interpolation of different binary 1D noise bands in one dimension along the other dimension. Let's again replace this with a an approach based on float A to make the gradients more varied.

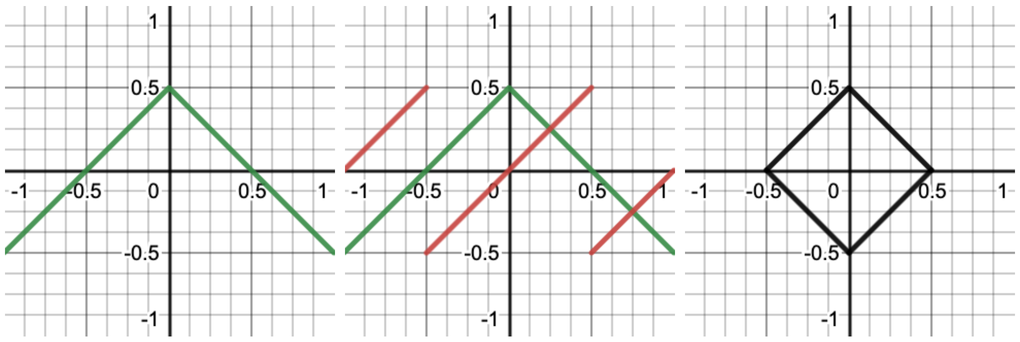
```
	public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y) =>
		(hash.Floats01A * 2f - 1f) * x;
```



Variable gradients along X.

In the case of 2D noise we are not limited to axis-aligned gradients, the gradient vector can rotate a full 360°. To generate such a vector we can use an approach similar to how we generate an octahedron-sphere shape, but reduced to two dimensions.

We start with a line along X, from −1 to 1. Then we make Y equal to 0.5 minus the absolute of X, like we defined Z when creating the first half of an octahedron, but now in one less dimension. This creates a wedge that can be considered an opened square. Then we subtract X plus 0.5 floored from X to shift the negative portions of the square so it is closed.

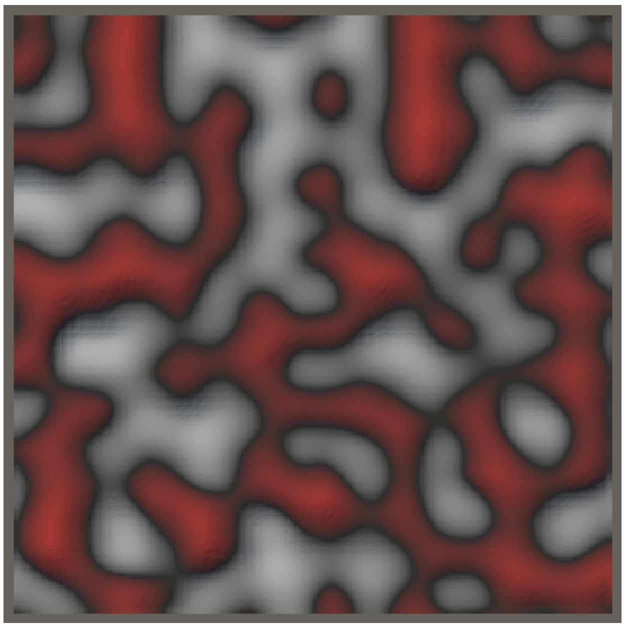


The process of creating a square from a line.

The result is a square with its corners on the X and Y axes at distance 0.5 from the origin. The gradient vectors thus point from the origin to somewhere on the edge of this square.

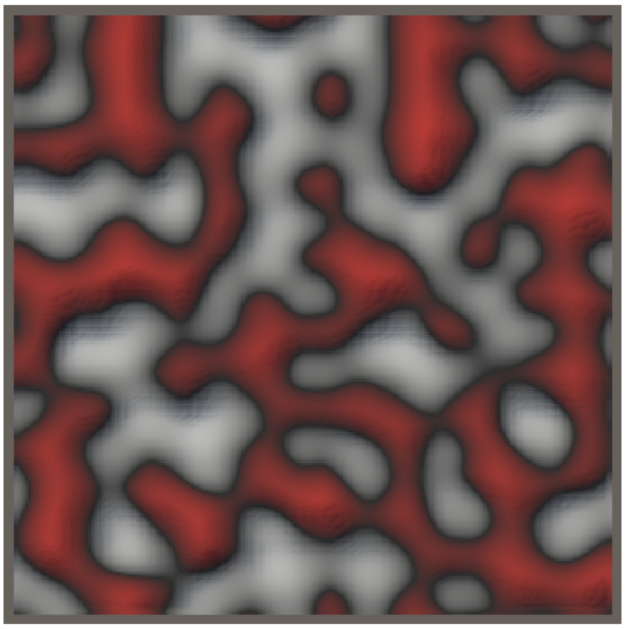
To evaluate the gradient in 2D we sum its X and Y components. Finally, we double the result so the axis-aligned vectors end up with a length of 1.

```
		public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y) {
			float4 gx = hash.Floats01A * 2f - 1f;
			float4 gy = 0.5f - abs(gx);
			gx -= floor(gx + 0.5f);
			return (gx * x + gy * y) * 2f;
		}
```


Square-based gradients.

If we want to turn the square distribution into a proper circle we can do so by normalizing the gradient vectors via division by their length. Note that this doesn't produce a uniform circular distribution of gradients, because they're uniformly distributed along the square instead. They're more concentrated near the square's corners, just like the point on the octahedron sphere are more concentrated near the corners of the octahedron. However this isn't a problem because the distribution is symmetrical and varied enough.

```
            return (gx * x + gy * y) * rsqrt(gx * gx + gy * gy);
```


Normalized gradients.

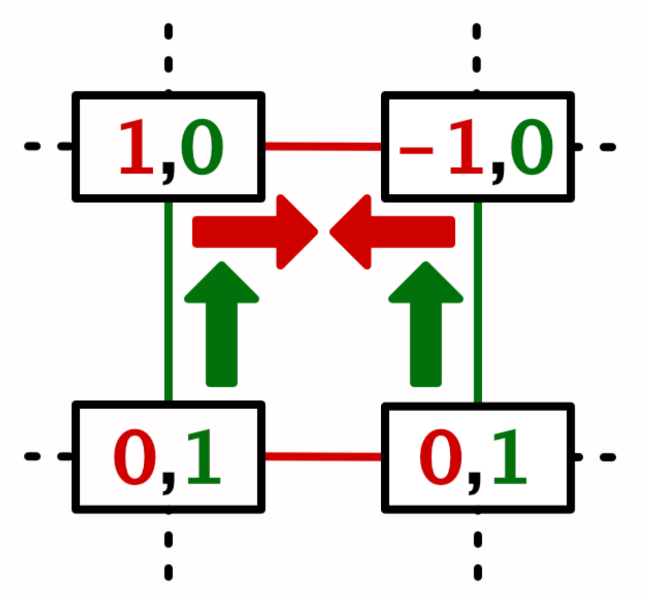
However, this doesn't make much difference so we can omit that part to make the noise code run faster. Normalization does make the diagonal gradients stronger, but the noise itself still needs to be normalized, which will also take care of that.

```
		return (gx * x + gy * y) * 2f;
```

2.5 Normalized 2D Noise

To normalize the noise we need to determine its current maximum amplitude. If a lattice square has four gradients that all point to its center then its maximum value will be reached in the middle. As that will be the average of four gradients that are equal, we only need to calculate the value of a single gradient at that point. A perfectly diagonal gradient is $f(x,y) = \frac{x+y}{2}$ so the amplitude at the middle would be $f(0.5, 0.5) = \frac{1}{2}$. However, this is not the maximum amplitude of the entire noise. There is another gradient configuration that has a greater amplitude.

The maximum in a single dimension is 0.5, in the middle of a lattice span when both gradients point straight toward each other along that axis. This is unnormalized 1D binary Perlin noise. If the two gradients of the other span of a lattice square both point straight at the opposite span, then during interpolation along the second dimension we might end up with a value that exceeds 0.5 somewhere.

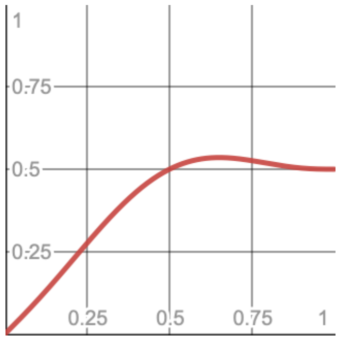


Gradients for maximum amplitude.

The maximum lies at the halfway point in the first dimension and somewhere along the interpolation in the second dimension. So this is effectively a smooth interpolation between an axis-aligned gradient and 0.5. Thus we have to find the maximum of the function $xs(1-x)+0.5s(x)$ with $s(t)=6t^5-15t^4+10t^3$ with 0–1 for the domain of x.

The expanded form of that function is $6x(1-x)^5-15x(1-x)^4+10x(1-x)^3+3x^5-7.5x^4+5x^3$, which simplifies to $-6x^6+18x^5-17.5x^4+5x^3+x$.

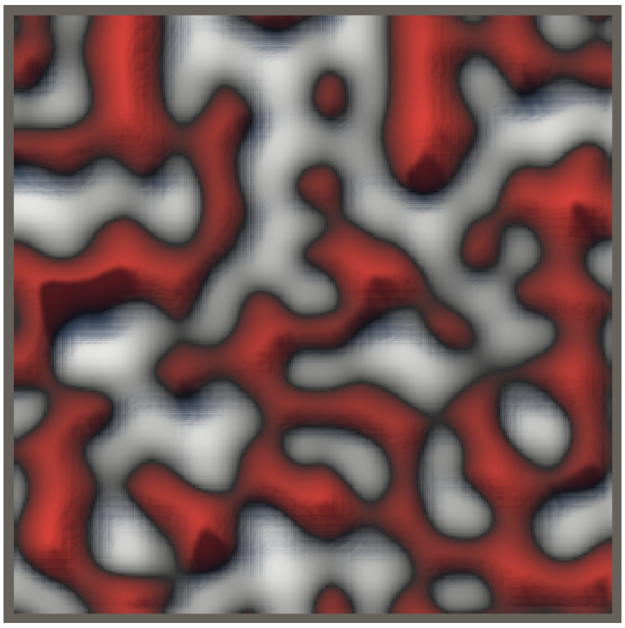
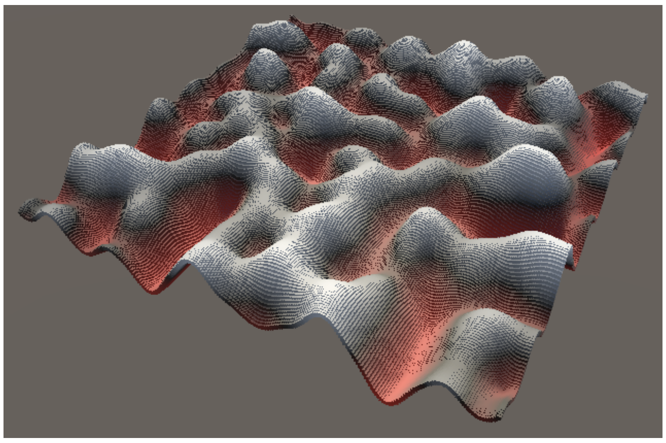
To find the maximum we have to take the derivative of that function and solve for zero, because where the rate of change is zero the function has reached a maximum or minimum. Solving that equation is possible but nontrivial. To get an idea of what the function looks like we can visualize it with Desmos. By selecting the graph line we can also immediately learn that the maximum is 0.5353 at X coordinate 0.6509. So it is indeed higher than 0.5.



2D maximum graph.

We can get a more precise answer by asking Wolfram|Alpha to maximize the function. It can even give us an exact answer but it is quite complex, so we make do with the approximation. 0.53528 will do, one digit more precise than what Desmos showed. Divide the gradient by that value to arrive at normalized 2D Perlin noise.

```
		return (gx * x + gy * y) * (2f / 0.53528f);
```

Normalized 2D Perlin noise.

What would be the maximum amplitude for circular gradients?

The circular version of a perfectly diagonal gradient is $f(x, y) = \frac{x+y}{\sqrt{2}}$. So the maximum amplitude found at the center of a lattice square with all gradients pointing at its center is $f(0.5, 0.5) = \frac{1}{\sqrt{2}} \approx 0.7071$. Thus to normalize the noise we'd have to divide by that, which is the same as multiplying with $\sqrt{2} \approx 1.4142$.

2.6 3D Gradients

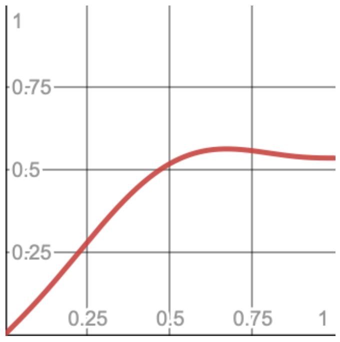
To create 3D Perlin noise we have to do the same thing that we did for the 2D version, expanded to include the third dimension. For this we can use the same octahedron-based approach that we used to generate the octahedron sphere. In this case we need two random values in the −1–1 range. We'll use floats A and D for this. D is preferred over B or C because a single bit shift is faster than both a shift and a mask operation.

```
        public float4 Evaluate (SmallXXHash4 hash, float4 x, float4 y, float4 z) {
            float4 gx = hash.Floats01A * 2f - 1f, gy = hash.Floats01D * 2f - 1f;
            float4 gz = 1f - abs(gx) - abs(gy);
            float4 offset = max(-gz, 0f);
            gx += select(-offset, offset, gx < 0f);
            gy += select(-offset, offset, gy < 0f);
            return (gx * x + gy * y + gz * z);
        }
```

Doesn't 3D Perlin noise use just twelve gradient vectors?

Perlin's reference implementation indeed picks from a set of twelve 2D diagonal gradient vectors with different orientations—pointing to the middle of the edges of a cube—with some repetition to get to a total of sixteen options. This way four bits can be converted to gradients. While this approach can be efficient for regular code or dedicated hardware, the nested binary branching isn't suitable for SIMD code. Our octahedron-based approach is both faster—if we don't normalize the gradients—and offers more variety.

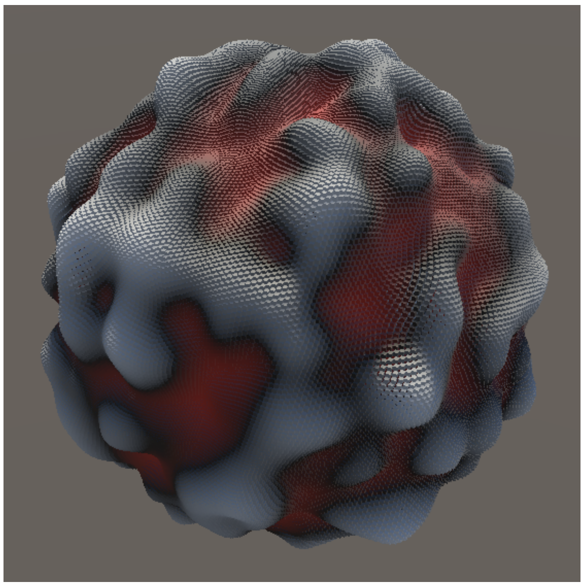
Finding the maximum amplitude works the same as for 2D. It is found while interpolating along the third dimension with a gradient that points along that axis, towards a 2D lattice rect that has achieved its maximum value, which is 0.53528. So we have to maximize the same function as before, except with the constant 0.5 replaced with 0.53528: $xs(1-x)+0.53528s(x)$, which can be expanded to $6x(1-x)^5-15x(1-x)^4+10x(1-x)^3+0.53528\left(6x^5-15x^4+10x^3\right)$. Feeding that to Desmos yields 0.5629 at X coordinate 0.6732.



3D maximum graph.

And Wolfram|Alpha tell us it is 0.56290 at 0.67321.

```
return (gx * x + gy * y + gz * z) * (1f / 0.56290f);
```



Octahedron-based 3D Perlin noise.

What would be the maximum amplitude for spherical 3D gradients?

For gradient vectors based on a sphere instead of an octahedron, the gradient vector pointing straight at the center of a lattice cube is $f(x, y, z) = \frac{x + y + z}{\sqrt{3}}$. So the maximum amplitude found in the middle of a lattice cube with all gradients pointing at its center is $f(0.5, 0.5, 0.5) = \frac{1.5}{\sqrt{3}} \approx 0.8660$. Thus to normalize the noise we'd have to divide by that.

The next tutorial is Noise Variants.