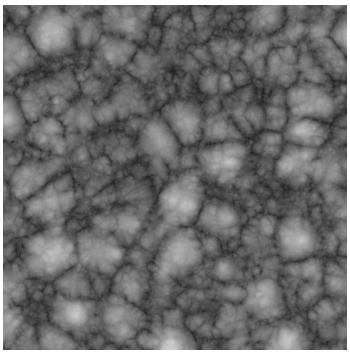
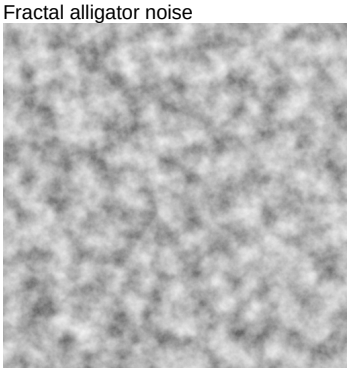
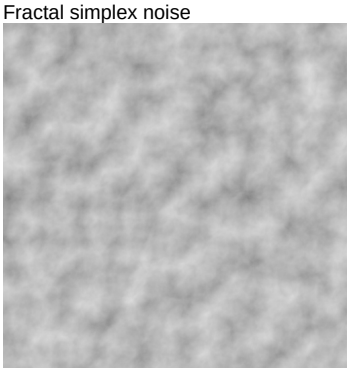
Additional procedural noises



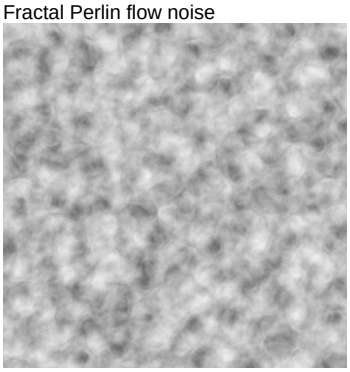
Fractal alligator noise



Fractal simplex noise



Fractal Perlin flow noise



Fractal Simplex flow noise

Page contents

Alligator noise

Overview

This is an implementation of the Houdini's Alligator noise as described here.

It uses the shading point's position in object space rather than the surface's texture coordinates.

Use this shader with a **VRayOSLTex** instance with the "***wrap texture coordinates***" option turned off.

The shader code can be found at alligatornoise.zip.

Parameters

start frequency - Initial sampling position multiplier that affects the overall granularity.

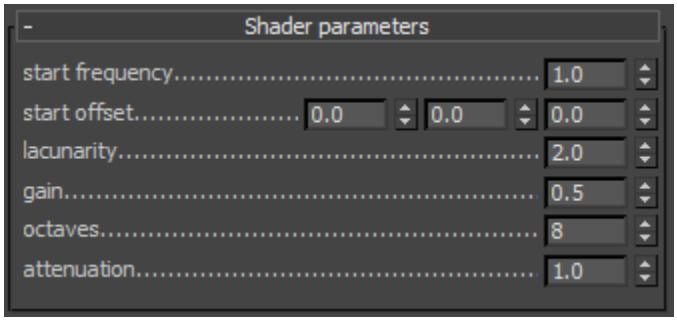
start offset - Offsets the initial sampling position effectively shifting the pattern in the specified direction.

lacunarity - Position (frequency) multiplier per iteration.

gain - Amplitude multiplier per iteration.

octaves - Number of fractal iterations.

attenuation - The power of the falloff applied to the final result.



Shader code

Here is the shader code:

alligatornoise.osl

```
/*
 * Copyright (c) 2016
 *        Side Effects Software Inc.  All rights reserved.
 *
 * Redistribution and use of Houdini Development Kit samples in source and
 * binary forms, with or without modification, are permitted provided that the
 * following conditions are met:
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 * 2. The name of Side Effects Software may not be used to endorse or
 *    promote products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY SIDE EFFECTS SOFTWARE `AS IS' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN
 * NO EVENT SHALL SIDE EFFECTS SOFTWARE BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
 * OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
 *
 *------------------------------------------------------------------------
 */

/// Alligator Noise is provided by Side Effects Software Inc. and is licensed
/// under a Creative Commons Attribution-ShareAlike 4.0 International License.
///
/// Author:  "Side Effects Software Inc"
/// Source:  "http://www.sidefx.com/docs/hdk15.0/alligator_2alligator_8_c-example.html"
/// License: "http://creativecommons.org/licenses/by-sa/4.0/"
///
/// Translated and modified by Ivan Mavrov, Chaos Group Ltd. 2016
/// Contact: ivan.mavrov@chaosgroup.com

/// 3D Alligator noise implementation.
/// Returned values are in the [0, 1] range.
float alligatorNoise3D(point position) {
        vector cellOffsets[27] = {
                vector( 0,  0,  0),
                vector( 1,  0,  0),
                vector( 1,  1,  0),
                vector( 0,  1,  0),
                vector(-1,  1,  0),
                vector(-1,  0,  0),
                vector(-1, -1,  0),
                vector( 0, -1,  0),
                vector( 1, -1,  0),

                vector( 0,  0, -1),
                vector( 1,  0, -1),
                vector( 1,  1, -1),
                vector( 0,  1, -1),
                vector(-1,  1, -1),
                vector(-1,  0, -1),
                vector(-1, -1, -1),
                vector( 0, -1, -1),
                vector( 1, -1, -1),

                vector( 0,  0,  1),
                vector( 1,  0,  1),
                vector( 1,  1,  1),
                vector( 0,  1,  1),
                vector(-1,  1,  1),
                vector(-1,  0,  1),
                vector(-1, -1,  1),
                vector( 0, -1,  1),
                vector( 1, -1,  1)
        };

        point iPosition = floor(position);

        float firstReverseSmoothPointDistance = 0.0;
        float secondReverseSmoothPointDistance = 0.0;

        for (int cellIndex = 0; cellIndex < 27; ++cellIndex) {
                point cellOrigin = iPosition + cellOffsets[cellIndex];
                vector cellPointOffset = cellnoise(cellOrigin, 0.0);
                point cellPointPosition = cellOrigin + cellPointOffset;

                float cellPointDistance = distance(position, cellPointPosition);

                if (cellPointDistance < 1.0) {
                        float reverseSmoothDistance = smoothstep(0.0, 1.0, 1.0 - cellPointDistance);

                        float distanceMultiplier = float(cellnoise(cellOrigin, 1.0));
                        reverseSmoothDistance *= distanceMultiplier;

                        if (firstReverseSmoothPointDistance < reverseSmoothDistance) {
                                secondReverseSmoothPointDistance = firstReverseSmoothPointDistance;
                                firstReverseSmoothPointDistance = reverseSmoothDistance;
                        } else {
```

```
                                if (secondReverseSmoothPointDistance < reverseSmoothDistance) {
                                        secondReverseSmoothPointDistance = reverseSmoothDistance;
                                }
                        }
                }
        }

        return firstReverseSmoothPointDistance - secondReverseSmoothPointDistance;
}

/// 3D Fractal Alligator noise implementation.
/// Returned values are in the [-1, 1] range.
float fractalAlligatorNoise3D(
        point position,
        float lacunarity, // Houdini 2.0
        float gain,       // Houdini rough
        int octaveCount   // Houdini turbulence - 1
) {
        float noiseValue = 0.0;

        float amplitude = 1.0;

    for (int octave = 0; octave < octaveCount; ++octave) {
                noiseValue += amplitude * (alligatorNoise3D(position) - 0.5);
                position *= lacunarity;
                amplitude *= gain;
        }

        return noiseValue;
}

shader FractalAlligatorNoise
        [[ string description = "Fractal Alligator noise" ]]
(
        float start_frequency = 1.0
                [[ string description = "Initial sampling position multiplier that affects the overall
granularity." ]],
        vector start_offset = vector(0.0)
                [[ string description = "Offsets the initial sampling position effectively shifting the pattern
in the specified direction." ]],

        float lacunarity = 2.0
                [[ string description = "Position (frequency) multiplier per iteration." ]],
        float gain = 0.5
                [[ string description = "Amplitude multiplier per iteration." ]],
        int octaves = 8
                [[ string description = "Number of fractal iterations." ]],

        float attenuation = 1.0
                [[ string description = "The power of the falloff applied to the final result." ]],

        output color result = 0.0
) {
        point objectPosition = transform("object", P);
        point startPosition = start_frequency * objectPosition - start_offset;

        float noiseValue = fractalAlligatorNoise3D(startPosition, lacunarity, gain, octaves);

        noiseValue = 0.5 * noiseValue + 0.5;
        noiseValue = pow(noiseValue, attenuation);

        result = color(noiseValue);
}
```

Simplex noise

Overview

Simplex noise is a close relative to the Perlin noise but with fewer directional artifacts and generally lower computational overhead especially in higher dimensions.

The current 3D implementation uses the shading point's position in object space rather than the surface's texture coordinates.

Use this shader with a **VRayOSLTex** instance with the "***wrap texture coordinates***" option turned off.

The shader code can be found at simplexnoise.zip.

Parameters

start frequency - Initial sampling position multiplier that affects the overall granularity.

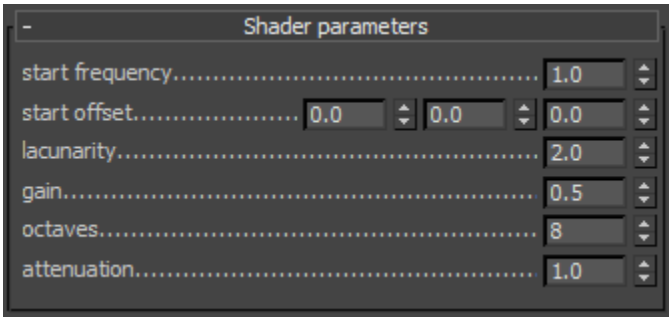
start offset - Offsets the initial sampling position effectively shifting the pattern in the specified direction.

lacunarity - Position (frequency) multiplier per iteration.

gain - Amplitude multiplier per iteration.

octaves - Number of fractal iterations.

attenuation - The power of the falloff applied to the final result.



Shader code

Here is the shader code:

simplexnoise.osl

```
/// 3D Simplex noise implementation.
/// Returned values are in the [0, 1] range.
float simplexNoise3D(point position) {
        int perm[512] = {
                151,160,137,91,90,15,
                131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
                190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
                88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
                77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
                102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
                135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
                5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
                223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
                129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
                251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
                49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
                138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180,
                151,160,137,91,90,15,
                131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
                190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
                88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
                77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
                102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
                135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
                5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
                223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
                129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
                251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
                49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
                138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
        };

        vector gradient[16] = {
                vector( 1.0,  1.0,  0.0),
                vector(-1.0,  1.0,  0.0),
                vector( 1.0, -1.0,  0.0),
                vector(-1.0, -1.0,  0.0),
                vector( 1.0,  0.0,  1.0),
                vector(-1.0,  0.0,  1.0),
```

```
        vector( 1.0,  0.0, -1.0),
        vector(-1.0,  0.0, -1.0),
        vector( 0.0,  1.0,  1.0),
        vector( 0.0, -1.0,  1.0),
        vector( 0.0,  1.0, -1.0),
        vector( 0.0, -1.0, -1.0),
        vector( 1.0,  1.0,  0.0),
        vector( 0.0, -1.0,  1.0),
        vector(-1.0,  1.0,  0.0),
        vector( 0.0, -1.0, -1.0)
};

float evaluatePointContribution(int i, int j, int k, vector offset) {
        float t = 0.6 - dot(offset, offset);
        if (t < 0.0) {
                return 0.0;
        }

        float t2 = t * t;
        float t4 = t2 * t2;

        int gradientIndex = perm[i + perm[j + perm[k]]];
        int safeGradientIndex = gradientIndex % 16;

        return t4 * dot(gradient[safeGradientIndex], offset);
}

// Skewing factors for the 3D simplex grid.
float F3 = 0.333333333;
float G3 = 0.166666667;
float G3a = 2.0 * G3;
float G3b = 3.0 * G3 - 1.0;

// Skew the input space to determine the simplex cell we are in.
float skew = (position[0] + position[1] + position[2]) * F3;
point skewedPosition = position + skew;
point skewedCellOrigin = floor(skewedPosition);

// Unskew the cell origin
float unskew = (skewedCellOrigin[0] + skewedCellOrigin[1] + skewedCellOrigin[2]) * G3;
point cellOrigin = skewedCellOrigin - unskew;

// The offset from the cell's origin a.k.a point 0
vector offset0 = position - cellOrigin;

// The second point offset from the cell origin in skewed space.
vector skewedOffset1;
// The third point offset from the cell origin in skewed space.
vector skewedOffset2;

if (offset0[0] >= offset0[1]) {
        if (offset0[1] >= offset0[2]) {
                // X Y Z order
                skewedOffset1 = vector(1, 0, 0);
                skewedOffset2 = vector(1, 1, 0);
        } else if (offset0[0] >= offset0[2]) {
                // X Z Y order
                skewedOffset1 = vector(1, 0, 0);
                skewedOffset2 = vector(1, 0, 1);
        } else {
                // Z X Y order
                skewedOffset1 = vector(0, 0, 1);
                skewedOffset2 = vector(1, 0, 1);
        }
} else {
        if (offset0[1] < offset0[2]) {
                // Z Y X order
                skewedOffset1 = vector(0, 0, 1);
                skewedOffset2 = vector(0, 1, 1);
        } else if (offset0[0] < offset0[2]) {
                // Y Z X order
```

```
                        skewedOffset1 = vector(0, 1, 0);
                        skewedOffset2 = vector(0, 1, 1);
                } else {
                        // Y X Z order
                        skewedOffset1 = vector(0, 1, 0);
                        skewedOffset2 = vector(1, 1, 0);
                }
        }

        // A step of (1, 0, 0) in skewed space means a step of (1 - G3, -G3, -G3) in regular space.
        // A step of (0, 1, 0) in skewed space means a step of (-G3, 1 - G3, -G3) in regular space.
        // A step of (0, 0, 1) in skewed space means a step of (-G3, -G3, 1 - G3) in regular space.

        // The offset from point 1 in regular space.
        vector offset1 = offset0 - skewedOffset1 + G3;

        // The offset from point 2 in regular space.
        vector offset2 = offset0 - skewedOffset2 + G3a;

        // The offset from point 3 in regular space.
        vector offset3 = offset0 + G3b;

        // Wrap the integer indices at 256, to avoid indexing perm[] out of bounds.
        int i = int(skewedCellOrigin[0]) & 255;
        int j = int(skewedCellOrigin[1]) & 255;
        int k = int(skewedCellOrigin[2]) & 255;

        // Accumulate the contributions from the four points.
        float noiseValue = 0.0;
        noiseValue += evaluatePointContribution(i, j, k, offset0);
        noiseValue += evaluatePointContribution(i + int(skewedOffset1[0]), j + int(skewedOffset1[1]), k + int
(skewedOffset1[2]), offset1);
        noiseValue += evaluatePointContribution(i + int(skewedOffset2[0]), j + int(skewedOffset2[1]), k + int
(skewedOffset2[2]), offset2);
        noiseValue += evaluatePointContribution(i + 1, j + 1, k + 1, offset3);

        // Scale to the [-1,1] range.
        noiseValue *= 32.0;

        // Scale to the [0, 1] range.
        return 0.5 * noiseValue + 0.5;
}

/// 3D Fractal Simplex noise implementation.
/// Returned values are in the [-1, 1] range.
float fractalSimplexNoise3D(
        point position,
        float lacunarity,
        float gain,
        int octaveCount
) {
        float noiseValue = 0.0;

        float amplitude = 1.0;

    for (int octave = 0; octave < octaveCount; ++octave) {
                noiseValue += amplitude * (simplexNoise3D(position) - 0.5);
                position *= lacunarity;
                amplitude *= gain;
        }

        return noiseValue;
}

shader FractalSimplexNoise
        [[ string description = "Fractal Simplex noise" ]]
(
        float start_frequency = 1.0
                [[ string description = "Initial sampling position multiplier that affects the overall
granularity." ]],
        vector start_offset = vector(0.0)
```

```
                [[ string description = "Offsets the initial sampling position effectively shifting the pattern
in the specified direction." ]],

        float lacunarity = 2.0
                [[ string description = "Position (frequency) multiplier per iteration." ]],
        float gain = 0.5
                [[ string description = "Amplitude multiplier per iteration." ]],
        int octaves = 8
                [[ string description = "Number of fractal iterations." ]],

        float attenuation = 1.0
                [[ string description = "The power of the falloff applied to the final result." ]],

        output color result = 0.0
) {
        point objectPosition = transform("object", P);
        point startPosition = start_frequency * objectPosition - start_offset;

        float noiseValue = fractalSimplexNoise3D(startPosition, lacunarity, gain, octaves);

        noiseValue = 0.5 * noiseValue + 0.5;
        noiseValue = pow(noiseValue, attenuation);

        result = color(noiseValue);
}
```

Perlin flow noise

Overview

It is a Perlin noise with an additional ***flow*** parameter that rotates the gradient vectors used to evaluate the noise.

This creates a 4-th dimension parameter with a period of 1.0 that morphs the noise rather than shifting it across the noise space.

This implementation uses Stefan Gustavson's concept of gradient rotation planes. You can learn more here.

It is a 3D noise that uses the shading point's position in object space rather than the surface's texture coordinates.

Use this shader with a **VRayOSLTex** instance with the "***wrap texture coordinates***" option turned off.

The shader code can be found at perlinflownoise.zip.

Parameters

start frequency - Initial sampling position multiplier that affects the overall granularity.

start offset - Offsets the initial sampling position effectively shifting the pattern in the specified direction.

flow - The coordinate of a special noise dimension with a period of 1 that naturally evolves the noise to animate it instead of sliding a 3D slice through the noise space.

lacunarity - Position (frequency) multiplier per iteration.

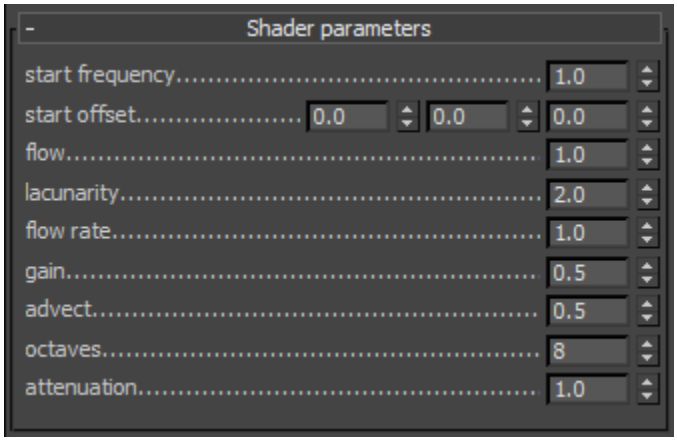
flow rate - Flow coordinate multiplier per iteration.

gain - Amplitude multiplier per iteration.

advect - Both initial advection amount and advection multiplier per iteration.

octaves - Number of fractal iterations.

attenuation - The power of the falloff applied to the final result.



Shader code

Here is the shader code:

perlinflownoise.osl

```
int fastFloor(float x) {
        return (x < 0) ? int(x) - 1 : int(x);
}

float fade(float t) {
        return t * t * t * (t * (t * 6.0 - 15.0) + 10.0);
}

float lerp(float t, float a, float b) {
        return a + t * (b - a);
}

/// 3D Perlin flow noise implementation.
/// Returned values are in the [0, 1] range.
float perlinFlowNoise3D(point position, float flow) {
        int perm[512] = {
                151,160,137,91,90,15,
                131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
                190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
                88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
                77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
                102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
                135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
                5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
                223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
                129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
                251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
                49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
                138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180,
                151,160,137,91,90,15,
                131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
                190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
                88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
                77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
                102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
                135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
                5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
                223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
                129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
                251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
                49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
                138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
        };

        // Gradient component that leads to a vector of length sqrt(2).
        // float a = sqrt(2)/sqrt(3);
        float a = 0.81649658;

        vector gradientUBase[16] = {
                vector( 1.0,  0.0,  1.0), vector( 0.0,  1.0,  1.0),
                vector(-1.0,  0.0,  1.0), vector( 0.0, -1.0,  1.0),
                vector( 1.0,  0.0, -1.0), vector( 0.0,  1.0, -1.0),
                vector(-1.0,  0.0, -1.0), vector( 0.0, -1.0, -1.0),
                vector( a,  a,  a), vector(-a,  a, -a),
                vector(-a, -a,  a), vector( a, -a, -a),
                vector(-a,  a,  a), vector( a, -a,  a),
                vector( a, -a, -a), vector(-a,  a, -a)
        };

        vector gradientVBase[16] = {
                vector(-a,  a,  a), vector(-a, -a,  a),
                vector( a, -a,  a), vector( a,  a,  a),
                vector(-a, -a, -a), vector( a, -a, -a),
                vector( a,  a, -a), vector(-a,  a, -a),
                vector( 1.0, -1.0,  0.0), vector( 1.0,  1.0,  0.0),
```

```
                vector(-1.0,  1.0,  0.0), vector(-1.0, -1.0,  0.0),
                vector( 1.0,  0.0,  1.0), vector(-1.0,  0.0,  1.0),
                vector( 0.0,  1.0, -1.0), vector( 0.0, -1.0, -1.0)
        };

        // Helper function to compute the rotated gradient.
        vector getGradient(int index, float sinTheta, float cosTheta) {
                int safeIndex = index % 16;
                vector gradientU = gradientUBase[safeIndex];
                vector gradientV = gradientVBase[safeIndex];
                return cosTheta * gradientU + sinTheta * gradientV;
        }

        float gradientDot(int index, float sinTheta, float cosTheta, float x, float y, float z) {
                vector gradient = getGradient(index, sinTheta, cosTheta);
                vector value = vector(x, y, z);
                return dot(gradient, value);
        }

        float x = position[0];
        float y = position[1];
        float z = position[2];

        int ix = fastFloor(x);
        int iy = fastFloor(y);
        int iz = fastFloor(z);

        float fx = x - float(ix);
        float fy = y - float(iy);
        float fz = z - float(iz);

        ix = ix & 255;
        iy = iy & 255;
        iz = iz & 255;

        float i = fade(fx);
        float j = fade(fy);
        float k = fade(fz);

        int A = perm[ix    ] + iy, AA = perm[A] + iz, AB = perm[A + 1] + iz;
        int B = perm[ix + 1] + iy, BA = perm[B] + iz, BB = perm[B + 1] + iz;

        // Sine and cosine for the gradient rotation angle
        float sinTheta = 0.0;
        float cosTheta = 0.0;
        sincos(M_2PI * flow, sinTheta, cosTheta);

        float noiseValue =
                lerp(k, lerp(j, lerp(i, gradientDot(perm[AA    ], sinTheta, cosTheta, fx      , fy      ,
fz      ),
                                                                gradientDot(perm[BA    ], sinTheta, cosTheta,
fx - 1.0, fy      , fz      )),
                                                lerp(i, gradientDot(perm[AB    ], sinTheta, cosTheta, fx      ,
fy - 1.0, fz      ),
                                                                gradientDot(perm[BB    ], sinTheta, cosTheta,
fx - 1.0, fy - 1.0, fz      ))),
                                        lerp(j, lerp(i, gradientDot(perm[AA + 1], sinTheta, cosTheta, fx      , fy
, fz - 1.0),
                                                                gradientDot(perm[BA + 1], sinTheta, cosTheta,
fx - 1.0, fy      , fz - 1.0)),
                                                lerp(i, gradientDot(perm[AB + 1], sinTheta, cosTheta, fx      ,
fy - 1.0, fz - 1.0),
                                                                gradientDot(perm[BB + 1], sinTheta, cosTheta,
fx - 1.0, fy - 1.0, fz - 1.0))));

        // Scale to the [0, 1] range.
        return 0.5 * noiseValue + 0.5;
}

vector perlinFlowNoise3DGradient(vector position, float flow, float delta) {
    vector result = vector(
```

```
                perlinFlowNoise3D(position + vector(delta, 0.0, 0.0), flow) -
                perlinFlowNoise3D(position - vector(delta, 0.0, 0.0), flow),
                perlinFlowNoise3D(position + vector(0.0, delta, 0.0), flow) -
                perlinFlowNoise3D(position - vector(0.0, delta, 0.0), flow),
                perlinFlowNoise3D(position + vector(0.0, 0.0, delta), flow) -
                perlinFlowNoise3D(position - vector(0.0, 0.0, delta), flow)
        );

    result /= (2.0 * delta);

    return result;
}

/// 3D Fractal Perlin flow noise implementation.
/// Returned values are in the [-1, 1] range.
float fractalPerlinFlowNoise3D(
        point position,
        float flow,
        float lacunarity,
        float flowRate,
        float gain,
        float advect,
        int octaveCount
) {
    float noiseValue = 0.0;

        float flowValue = flow;

        float amplitude = 1.0;
        float advectionAmount = advect;

        for (int octave = 0; octave < octaveCount; ++octave) {
                float noiseOctave = amplitude * (perlinFlowNoise3D(position, flowValue) - 0.5);
                noiseValue += noiseOctave;

                if (advectionAmount != 0.0) {
                        position -= advectionAmount * noiseOctave * perlinFlowNoise3DGradient(position, flow,
0.01);
                }

                position *= lacunarity;
                flowValue *= flowRate;
                amplitude *= gain;
                advectionAmount *= advect;
        }

        return noiseValue;
}

shader FractalPerlinFlowNoise
        [[ string description = "Fractal Perlin flow noise" ]]
(
        float start_frequency = 1.0
                [[ string description = "Initial sampling position multiplier that affects the overall
granularity." ]],
        vector start_offset = vector(0.0)
                [[ string description = "Offsets the initial sampling position effectively shifting the pattern
in the specified direction." ]],

        float flow = 1.0
                [[ string description = "The coordinate of a special noise dimension with a period of 1 that
naturally evolves the noise to animate it instead of sliding a 3D slice throught the noise space." ]],

        float lacunarity = 2.0
                [[ string description = "Position (frequency) multiplier per iteration." ]],
        float flow_rate = 1.0
                [[ string description = "Flow coordinate multiplier per iteration." ]],
        float gain = 0.5
                [[ string description = "Amplitude multiplier per iteration." ]],
        float advect = 0.5
                [[ string description = "Both initial advection amount and advection multiplier per iteration."
```

```
]],
        int octaves = 8
                [[ string description = "Number of fractal iterations." ]],

        float attenuation = 1.0
                [[ string description = "The power of the falloff applied to the final result." ]],

        output color result = 0.0
) {
        point objectPosition = transform("object", P);
        point startPosition = start_frequency * objectPosition - start_offset;

        float noiseValue = fractalPerlinFlowNoise3D(startPosition, flow, lacunarity, flow_rate, gain, advect,
octaves);

        noiseValue = 0.5 * noiseValue + 0.5;
        noiseValue = pow(noiseValue, attenuation);

        result = color(noiseValue);
}
```

Simplex flow noise

Overview

This is an OSL translation of Stefan Gustavson's Simplex flow noise implementation with fractal noise extension.

The Simplex flow noise uses true analytic derivatives that give a significant performance edge over the Perlin flow noise.

It is a 3D noise that uses the shading point's position in object space rather than the surface's texture coordinates.

Use this shader with a **VRayOSLTex** instance with the "***wrap texture coordinates***" option turned off.

The shader code can be found at simplexflownoise.zip.

Parameters

start frequency - Initial sampling position multiplier that affects the overall granularity.

start offset - Offsets the initial sampling position effectively shifting the pattern in the specified direction.

flow - The coordinate of a special noise dimension with a period of 1 that naturally evolves the noise to animate it instead of sliding a 3D slice through the noise space.

lacunarity - Position (frequency) multiplier per iteration.

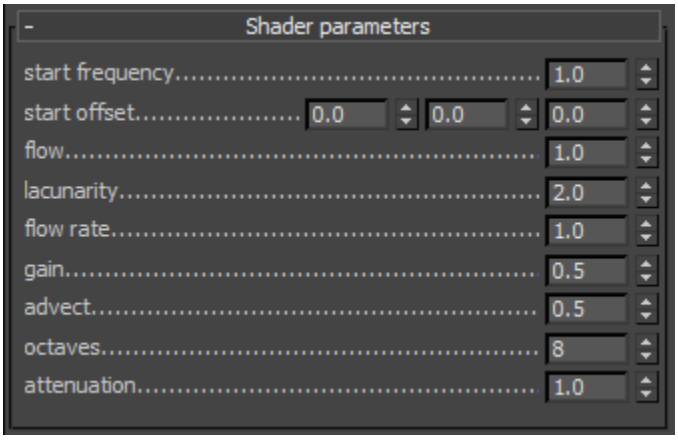
flow rate - Flow coordinate multiplier per iteration.

gain - Amplitude multiplier per iteration.

advect - Both initial advection amount and advection multiplier per iteration.

octaves - Number of fractal iterations.

attenuation - The power of the falloff applied to the final result.



Shader code

Here is the shader code:

simplexflownoise.osl

```
/// Oiginal notice:
```

```
/*
 * srdnoise23, Simplex noise with rotating gradients
 * and a true analytic derivative in 2D and 3D.
 *
 * This is version 2 of srdnoise23 written in early 2008.
 * A stupid bug was corrected. Do not use earlier versions.
 *
 * Author: Stefan Gustavson, 2003-2008
 *
 * Contact: stefan.gustavson@gmail.com
 *
 * This code was GPL licensed until February 2011.
 * As the original author of this code, I hereby
 * release it into the public domain.
 * Please feel free to use it for whatever you want.
 * Credit is appreciated where appropriate, and I also
 * appreciate being told where this code finds any use,
 * but you may do as you like.
 */

/// Translated and modified by Ivan Mavrov, Chaos Group Ltd. 2016
/// Contact: ivan.mavrov@chaosgroup.com

/// 3D Simplex flow noise implementation.
/// Returned values are in the [0, 1] range.
float simplexFlowNoise3D(point position, float flow, output vector dNoise) {
        int perm[512] = {
                151,160,137,91,90,15,
                131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
                190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
                88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
                77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
                102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
                135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
                5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
                223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
                129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
                251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
                49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
                138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180,
                151,160,137,91,90,15,
                131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
                190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
                88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
                77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
                102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
                135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
                5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
                223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
                129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
                251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
                49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
                138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
        };

        // Gradient component that leads to a vector of length sqrt(2).
        // float a = sqrt(2)/sqrt(3);
        float a = 0.81649658;

        vector gradientUBase[16] = {
                vector( 1.0,  0.0,  1.0), vector( 0.0,  1.0,  1.0),
                vector(-1.0,  0.0,  1.0), vector( 0.0, -1.0,  1.0),
                vector( 1.0,  0.0, -1.0), vector( 0.0,  1.0, -1.0),
                vector(-1.0,  0.0, -1.0), vector( 0.0, -1.0, -1.0),
                vector( a,  a,  a), vector(-a,  a, -a),
                vector(-a, -a,  a), vector( a, -a, -a),
                vector(-a,  a,  a), vector( a, -a,  a),
                vector( a, -a, -a), vector(-a,  a, -a)
        };
```

```
vector gradientVBase[16] = {
        vector(-a,  a,  a), vector(-a, -a,  a),
        vector( a, -a,  a), vector( a,  a,  a),
        vector(-a, -a, -a), vector( a, -a, -a),
        vector( a,  a, -a), vector(-a,  a, -a),
        vector( 1.0, -1.0,  0.0), vector( 1.0,  1.0,  0.0),
        vector(-1.0,  1.0,  0.0), vector(-1.0, -1.0,  0.0),
        vector( 1.0,  0.0,  1.0), vector(-1.0,  0.0,  1.0),
        vector( 0.0,  1.0, -1.0), vector( 0.0, -1.0, -1.0)
};

// Helper function to compute the rotated gradient.
vector getGradient(int index, float sinTheta, float cosTheta) {
        int safeIndex = index % 16;
        vector gradientU = gradientUBase[safeIndex];
        vector gradientV = gradientVBase[safeIndex];
        return cosTheta * gradientU + sinTheta * gradientV;
}

// Skewing factors for the 3D simplex grid.
// float F3 = 1.0 / 3.0;
// float G3 = 1.0 / 6.0;
float F3 = 0.333333333;
float G3 = 0.166666667;
float G3a = 2.0 * G3;
float G3b = 3.0 * G3 - 1.0;

// Skew the input space to determine the simplex cell we are in.
float skew = (position[0] + position[1] + position[2]) * F3;
point skewedPosition = position + skew;
point skewedCellOrigin = floor(skewedPosition);

// Unskew the cell origin
float unskew = (skewedCellOrigin[0] + skewedCellOrigin[1] + skewedCellOrigin[2]) * G3;
point cellOrigin = skewedCellOrigin - unskew;

// The offset from the cell's origin a.k.a point 0
vector offset0 = position - cellOrigin;

// The second point offset from the cell origin in skewed space.
vector skewedOffset1;
// The third point offset from the cell origin in skewed space.
vector skewedOffset2;

if (offset0[0] >= offset0[1]) {
        if (offset0[1] >= offset0[2]) {
                // X Y Z order
                skewedOffset1 = vector(1, 0, 0);
                skewedOffset2 = vector(1, 1, 0);
        } else if (offset0[0] >= offset0[2]) {
                // X Z Y order
                skewedOffset1 = vector(1, 0, 0);
                skewedOffset2 = vector(1, 0, 1);
        } else {
                // Z X Y order
                skewedOffset1 = vector(0, 0, 1);
                skewedOffset2 = vector(1, 0, 1);
        }
} else {
        if (offset0[1] < offset0[2]) {
                // Z Y X order
                skewedOffset1 = vector(0, 0, 1);
                skewedOffset2 = vector(0, 1, 1);
        } else if (offset0[0] < offset0[2]) {
                // Y Z X order
                skewedOffset1 = vector(0, 1, 0);
                skewedOffset2 = vector(0, 1, 1);
        } else {
                // Y X Z order
                skewedOffset1 = vector(0, 1, 0);
                skewedOffset2 = vector(1, 1, 0);
```

```
                }
        }

        // A step of (1, 0, 0) in skewed space means a step of (1 - G3, -G3, -G3) in regular space.
        // A step of (0, 1, 0) in skewed space means a step of (-G3, 1 - G3, -G3) in regular space.
        // A step of (0, 0, 1) in skewed space means a step of (-G3, -G3, 1 - G3) in regular space.

        // The offset from point 1 in regular space.
        vector offset1 = offset0 - skewedOffset1 + G3;

        // The offset from point 2 in regular space.
        vector offset2 = offset0 - skewedOffset2 + G3a;

        // The offset from point 3 in regular space.
        vector offset3 = offset0 + G3b;

        // Wrap the integer indices at 256, to avoid indexing perm[] out of bounds.
        int i = int(skewedCellOrigin[0]) & 255;
        int j = int(skewedCellOrigin[1]) & 255;
        int k = int(skewedCellOrigin[2]) & 255;

        // Sine and cosine for the gradient rotation angle.
        float sinTheta = 0.0;
        float cosTheta = 0.0;
        sincos(M_2PI * flow, sinTheta, cosTheta);

        // Calculate the contribution from the four points.
        float t0 = 0.6 - dot(offset0, offset0);
        float t02 = 0.0;
        float t04 = 0.0;
        vector gradient0 = vector(0.0);
        float n0 = 0.0;
        if (t0 < 0.0) {
                t0 = 0.0;
        } else {
                t02 = t0 * t0;
                t04 = t02 * t02;
                gradient0 = getGradient(perm[i + perm[j + perm[k]]], sinTheta, cosTheta);
                n0 = t04 * dot(gradient0, offset0);
        }

        float t1 = 0.6 - dot(offset1, offset1);
        float t12 = 0.0;
        float t14 = 0.0;
        vector gradient1 = vector(0.0);
        float n1 = 0.0;
        if (t1 < 0.0) {
                t1 = 0.0;
        } else {
                t12 = t1 * t1;
                t14 = t12 * t12;
                gradient1 = getGradient(perm[i + int(skewedOffset1[0]) + perm[j + int(skewedOffset1[1]) + perm
[k + int(skewedOffset1[2])]]], sinTheta, cosTheta);
                n1 = t14 * dot(gradient1, offset1);
        }

        float t2 = 0.6 - dot(offset2, offset2);
        float t22 = 0.0;
        float t24 = 0.0;
        vector gradient2 = vector(0.0);
        float n2 = 0.0;
        if(t2 < 0.0) {
                t2 = 0.0;
        } else {
                t22 = t2 * t2;
                t24 = t22 * t22;
                gradient2 = getGradient(perm[i + int(skewedOffset2[0]) + perm[j + int(skewedOffset2[1]) + perm
[k + int(skewedOffset2[2])]]], sinTheta, cosTheta);
                n2 = t24 * dot(gradient2, offset2);
        }
```

```
        float t3 = 0.6 - dot(offset3, offset3);
        float t32 = 0.0;
        float t34 = 0.0;
        vector gradient3 = vector(0.0);
        float n3 = 0.0;
        if(t3 < 0.0) {
                t32 = 0.0;
        } else {
                t32 = t3 * t3;
                t34 = t32 * t32;
                gradient3 = getGradient(perm[i + 1 + perm[j + 1 + perm[k + 1]]], sinTheta, cosTheta);
                n3 = t34 * dot(gradient3, offset3);
        }

        // Accumulate the contributions from each point to get the final noise value.
        // The result is scaled to return values in the range [-1,1].
        float noiseValue = 32.0 * (n0 + n1 + n2 + n3);

        // Compute noise derivative.
        //  *dnoise_dx = -8.0f * t02 * t0 * x0 * dot(gx0, gy0, gz0, x0, y0, z0) + t04 * gx0;
        //  *dnoise_dy = -8.0f * t02 * t0 * y0 * dot(gx0, gy0, gz0, x0, y0, z0) + t04 * gy0;
        //  *dnoise_dz = -8.0f * t02 * t0 * z0 * dot(gx0, gy0, gz0, x0, y0, z0) + t04 * gz0;
        //  *dnoise_dx += -8.0f * t12 * t1 * x1 * dot(gx1, gy1, gz1, x1, y1, z1) + t14 * gx1;
        //  *dnoise_dy += -8.0f * t12 * t1 * y1 * dot(gx1, gy1, gz1, x1, y1, z1) + t14 * gy1;
        //  *dnoise_dz += -8.0f * t12 * t1 * z1 * dot(gx1, gy1, gz1, x1, y1, z1) + t14 * gz1;
        //  *dnoise_dx += -8.0f * t22 * t2 * x2 * dot(gx2, gy2, gz2, x2, y2, z2) + t24 * gx2;
        //  *dnoise_dy += -8.0f * t22 * t2 * y2 * dot(gx2, gy2, gz2, x2, y2, z2) + t24 * gy2;
        //  *dnoise_dz += -8.0f * t22 * t2 * z2 * dot(gx2, gy2, gz2, x2, y2, z2) + t24 * gz2;
        //  *dnoise_dx += -8.0f * t32 * t3 * x3 * dot(gx3, gy3, gz3, x3, y3, z3) + t34 * gx3;
        //  *dnoise_dy += -8.0f * t32 * t3 * y3 * dot(gx3, gy3, gz3, x3, y3, z3) + t34 * gy3;
        //  *dnoise_dz += -8.0f * t32 * t3 * z3 * dot(gx3, gy3, gz3, x3, y3, z3) + t34 * gz3;

        dNoise  = t02 * t0 * offset0 * dot(gradient0, offset0) + t04 * gradient0;
        dNoise += t12 * t1 * offset1 * dot(gradient1, offset1) + t14 * gradient1;
        dNoise += t22 * t2 * offset2 * dot(gradient2, offset2) + t24 * gradient2;
        dNoise += t32 * t3 * offset3 * dot(gradient3, offset3) + t34 * gradient3;
        dNoise *= -8.0;

        // Scale noise derivative to match the noise scaling.
        //dNoise *= (32.0 / 2.0);

        // Scale to the [0, 1] range.
        return 0.5 * noiseValue + 0.5;
}

/// 3D Fractal Simplex flow noise implementation.
/// Returned values are in the [-1, 1] range.
float fractalSimplexFlowNoise3D(
        point position,
        float flow,
        float lacunarity,
        float flowRate,
        float gain,
        float advect,
        int octaveCount
) {
        float noiseValue = 0.0;

        float flowValue = flow;

        float amplitude = 1.0;
        float advectionAmount = advect;

        for (int octave = 0; octave < octaveCount; ++octave) {
                vector flownoise3DGradient = vector(0.0);
                float noiseOctave = amplitude * (simplexFlowNoise3D(position, flowValue, flownoise3DGradient) -
0.5);
                noiseValue += noiseOctave;

                position -= advectionAmount * noiseOctave * flownoise3DGradient;
```

```
                position *= lacunarity;
                flowValue *= flowRate;
                amplitude *= gain;
                advectionAmount *= advect;
        }

        return noiseValue;
}

shader FractalSimplexFlowNoise
        [[ string description = "Fractal Simplex flow noise" ]]
(
        float start_frequency = 1.0
                [[ string description = "Initial sampling position multiplier that affects the overall
granularity." ]],
        vector start_offset = vector(0.0)
                [[ string description = "Offsets the initial sampling position effectively shifting the pattern
in the specified direction." ]],

        float flow = 1.0
                [[ string description = "The coordinate of a special noise dimension with a period of 1 that
naturally evolves the noise to animate it instead of sliding a 3D slice throught the noise space." ]],

        float lacunarity = 2.0
                [[ string description = "Position (frequency) multiplier per iteration." ]],
        float flow_rate = 1.0
                [[ string description = "Flow coordinate multiplier per iteration." ]],
        float gain = 0.5
                [[ string description = "Amplitude multiplier per iteration." ]],
        float advect = 0.5
                [[ string description = "Both initial advection amount and advection multiplier per iteration."
]],
        int octaves = 8
                [[ string description = "Number of fractal iterations." ]],

        float attenuation = 1.0
                [[ string description = "The power of the falloff applied to the final result." ]],

        output color result = 0.0
) {
        point objectPosition = transform("object", P);
        point startPosition = start_frequency * objectPosition - start_offset;

        float noiseValue = fractalSimplexFlowNoise3D(startPosition, flow, lacunarity, flow_rate, gain, advect,
octaves);

        noiseValue = 0.5 * noiseValue + 0.5;
        noiseValue = pow(noiseValue, attenuation);

        result = color(noiseValue);
}
```