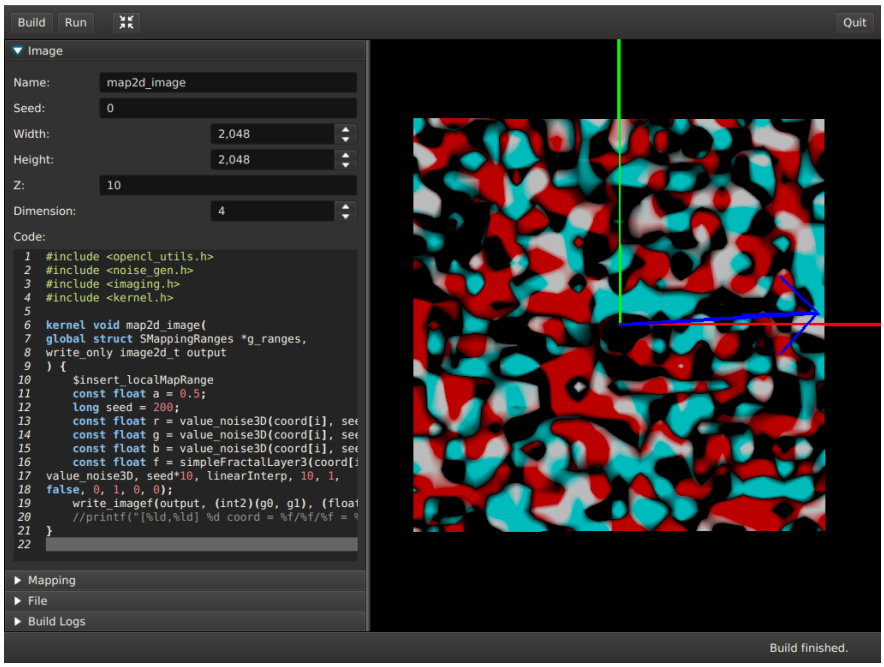
# ANL-OpenCL

## Documentation

Erwin Müller

November 25, 2021

*https://anl-opencl.anrisoftware.com/*




The purpose of this document is to describe the *ANL-OpenCL* library, how to use it to create noise images and how to use the bundled app.

# Contents

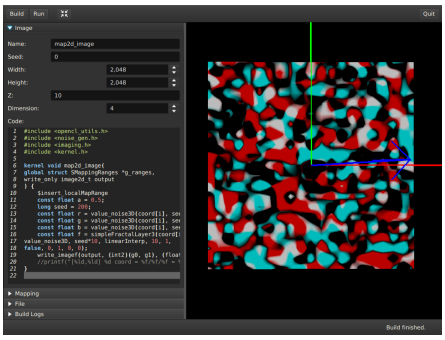


Figure 1: Screenshot of the app version 0.0.2

# 1 App

The bundled app is a graphical user interface to enter the kernel code, build it and generate a preview of the noise image. It is implemented in Java and JMonkeyEngine 3. The goal of the app is to quickly prototype noise images with kernel code. The goal is not to provide an IDE or an advanced code editor. The app is divided into two parts. One part is to enter the kernel code and parameters and other other part is to display the generated images.

## 1.1 Toolbar

Figure 2: Toolbar

The toolbar have buttons for the most common functions.

### 1.1.1 Build

Figure 3: Toolbar Build Button

The Build button will build the kernel code and display the generated image according to the parameters.

### 1.1.2 Reset Camera

Figure 4: Toolbar Build Button

The Reset Camera button will reset the camera to the default position and zoom. Key shortcut F10.

### 1.1.3 Quit

Figure 5: Toolbar Build Button

The Quit button will exit the app. Key shortcut Ctrl-Q.

## 1.2 Image

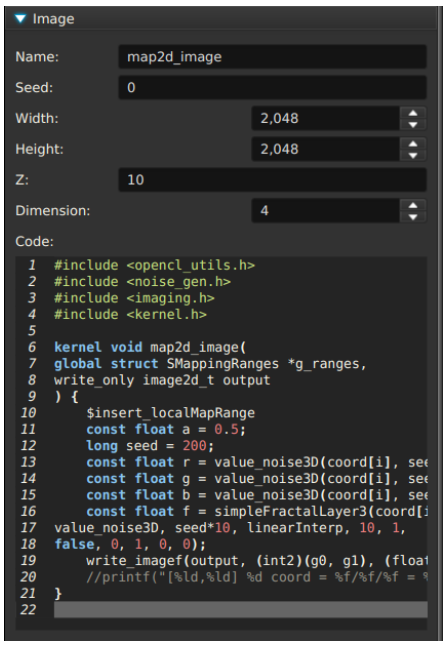



Figure 6: Image

The image window have the image parameters and the kernel code.

#### 1.2.1 Name

The name of the kernel to build.

#### 1.2.2 Seed

The seed number.

#### 1.2.3 Width

The width of the image in pixels.

#### 1.2.4 Height

The height of the image in pixels.

#### 1.2.5 Z

The Z value.

#### 1.2.6 Dimension

The count of float numbers of each coordinate. Different noise functions expect to have the correct dimension of the coordinates available.

**2D** requires dimension of 2 for `float2`;

**3D** requires dimension of 4 for `float3`;

**4D** requires dimension of 4 for `float4`;

**6D** requires dimension of 8 for `float8`;

## 1.3 Mapping

The mapping window have the parameters to map coordinates.

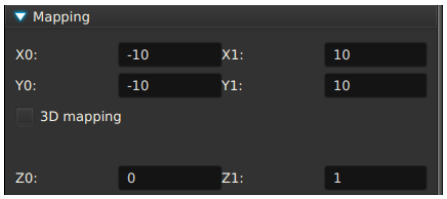



Figure 7: Image

#### 1.3.1 3D mapping

If enabled then the mapping is done in 3D and the function `map3D` must be used.

## 1.4 Scene Window

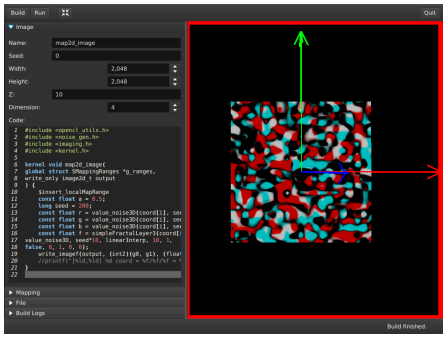


Figure 8: Image

The scene window shows the generated images. The scene can be moved and zoomed with the mouse.

**Middle Mouse** moving the scene;

**Mouse Wheel** zooming the scene;

# 2 API

This section documents the C functions and types that can be used in *ANL-OpenCL*.

## 2.1 OpenCL Kernel

The basic structure of a kernel is give as an example in listings 1. The kernel expects two parameters to generale an image as a texture. The first are the `SMappingRanges` that contains the parameters of the mapping as entered by the user on the Mapping 1.2.6 window. The second parameter is the `image2d_t` image output.

The kernel code can contain variables that are inserted before the code is build. One of those variables is `$insert_localMapRange`. This variable is replaced with the code in listings 2. `$insert_localMapRange` contains code that divides the coordinates space given in the mapping ranges into small peaces and allows the noise functions to operate in the local group on the divided part. After this code was inserted the `const int i` variable contains the current index in the `vector3 coord` variable, i.e. reading the coordinate from the position `coord[i]` will return the correct coordinate in the local group. Additionally, the variable `struct SMappingRanges ranges` will contain the local mapping ranges.

The variable `$localSize` contains the size of the part of the image that we want to process in pixels. Currently it is set to a maximum size of 32. That means that an image of the size 1024x1024 is divided into 32x32 ($1024/32 = 32$) parts with the size of 32x32 pixels. Each noise function will be called in the local group only on the 32x32 pixels part.

The variable `$z` contains the Z value from the Image 1.1.3 section.

```
#include <opencl_utils.h>
#include <noise_gen.h>
#include <imaging.h>
#include <kernel.h>

kernel void map2d_image(
global struct SMappingRanges *g_ranges,
write_only image2d_t output
) {
    $insert_localMapRange
    const float a = 0.5;
    long seed = 200;
    const float r = value_noise3D(coord[i], seed, linearInterp);
    const float g = value_noise3D(coord[i], seed*2, linearInterp);
    const float b = value_noise3D(coord[i], seed*2, linearInterp);
    const float f = simpleFractalLayer3(coord[i], value_noise3D, seed*10, linearInterp, 10, 1, false, 0, 1, 0, 0);
    write_imagef(output, (int2)(g0, g1), (float4)(r*f, g*f, b*f, a));
}
```

Listing 1: Kernel Example

```
const size_t g0 = get_global_id(0);
const size_t g1 = get_global_id(1);
const size_t w = get_global_size(0);
const size_t h = get_global_size(1);
const size_t l0 = get_local_id(0);
const size_t l1 = get_local_id(1);
const size_t lw = get_local_size(0);
const size_t lh = get_local_size(1);
local vector3 coord[$localSize * $localSize];
local struct SMappingRanges ranges;
if (l0 == 0 && l1 == 0) {
    const REAL sw = (g_ranges->mapx1 - g_ranges->mapx0) / w;
    const REAL sh = (g_ranges->mapy1 - g_ranges->mapy0) / h;
    const REAL x0 = g_ranges->mapx0 + g0 * sw;
    const REAL x1 = g_ranges->mapx0 + g0 * sw + sw * lw;
    const REAL y0 = g_ranges->mapy0 + g1 * sh;
    const REAL y1 = g_ranges->mapy0 + g1 * sh + sh * lh;
    set_ranges_map2D(&ranges, x0, x1, y0, y1);
    map2D(coord, calc_seamless_none, ranges, lw, lh, $z);
}
work_group_barrier(CLK_LOCAL_MEM_FENCE);
const int i = (l0 + l1 * lh);
```

Listing 2: Kernel Example

## 2.2 Basic Types

*ANL-OpenCL* is designed to be used with both `float` and `double` floating point types. To use double floating point types the flag `ANLOPENCL_USE_DOUBLE` must be set.

The header file `opencl_utils.h` defines the custom types

- `vector2`
- `vector3`
- `vector4`
- `vector8`
- `vector16`

as either `float2`, `float3`, `float4`, `float8`, `float16` or `double2`, `double3`, `double4`, `double8`, `double16`. And the `REAL` type as either `float` or `double`.

## 2.3 Mapping Ranges Type

Since *ANL-OpenCL* is a port of the Josua Tippetts' C++ library http://accidentalnoise.sourceforge.net/index.html the same documentation can be used for the mapping ranges.

The struct `SMappingRanges` is defined with the fields

- `mapx0`, `mapy0`, `mapz0`, `mapx1`, `mapy1`, `mapz1` and
- `loopx0`, `loopy0`, `loopz0`, `loopx1`, `loopy1`, `loopz1`

## 2.4 Mapping Ranges Functions

### 2.4.1 Create Mapping Ranges Functions

```
struct SMappingRanges
    create_ranges_default();
struct SMappingRanges
    create_ranges_map2D(REAL x0, REAL
    x1, REAL y0, REAL y1);
struct SMappingRanges
    create_ranges_map3D(REAL x0, REAL
    x1, REAL y0, REAL y1, REAL z0,
    REAL z1)
```

Listing 3: Definition of create ranges functions

```
struct SMappingRanges ranges =
    create_ranges_default();
//
struct SMappingRanges ranges =
    create_ranges_map2D(-10, 10, -10,
    10);
//
struct SMappingRanges ranges =
    create_ranges_map3D(-10, 10, -10,
    10, 0, 1);
```

Listing 4: Example for create ranges functions

Defined in `imaging.h`. The first function creates a default `SMappingRanges` from -1 to 1. The other functions create `SMappingRanges` with the specified ranges.

### 2.4.2 Set Mapping Ranges Functions

```
struct SMappingRanges
    set_ranges_default(struct
    SMappingRanges *const ranges);
struct SMappingRanges set_ranges_map2D
    (struct SMappingRanges *const
    ranges, REAL x0, REAL x1, REAL y0,
     REAL y1);
struct SMappingRanges set_ranges_map3D
    (struct SMappingRanges *const
    ranges, REAL x0, REAL x1, REAL y0,
     REAL y1, REAL z0, REAL z1);
```

Listing 5: Definition of set ranges functions

```
struct SMappingRanges ranges;
set_ranges_default(&ranges);
//
set_ranges_map2D(&ranges, -10, 10,
    -10, 10);
//
set_ranges_map3D(&ranges, -10, 10,
    -10, 10, 0, 1);
```

Listing 6: Example for set ranges functions

Defined in `imaging.h`. The first function sets a default `SMappingRanges` from -1 to 1. The other functions sets `SMappingRanges` with the specified ranges.

### 2.4.3 Mapping Functions

```
void* map2D(void *out, calc_seamless
    calc_seamless, struct
    SMappingRanges ranges, size_t
    width, size_t height, REAL z);
void* map2DNoZ(void *out,
    calc_seamless_no_z calc_seamless,
    struct SMappingRanges ranges,
    size_t width, size_t height);
```

Listing 7: Definition of mapping functions

```
struct SMappingRanges ranges =
    create_ranges_map2D(-10, 10, -10,
    10);
map2D(coord, calc_seamless_none,
    ranges, 1024, 1024, 0);
```

Listing 8: Example for mapping functions

Defined in `imaging.h`. Creates the mapping ranges for the width and height. Different seamless functions are supported. The coordinates must have the correct size that is dependent on the seamless function used. For the `map2D` function the following seamless functions are available and the coordinates type must be used:

- `calc_seamless_none` coordinates must be of `vector3`
- `calc_seamless_x` coordinates must be of `vector4`
- `calc_seamless_y` coordinates must be of `vector4`
- `calc_seamless_z` coordinates must be of `vector4`
- `calc_seamless_xy` coordinates must be of `vector8`
- `calc_seamless_xz` coordinates must be of `vector8`
- `calc_seamless_yz` coordinates must be of `vector8`
- `calc_seamless_xyz` coordinates must be of `vector8`

For the `map2DNoZ` function the following seamless functions are available and the coordinates type must be used:

- `calc_seamless_no_z_none` coordinates must be of `vector2`
- `calc_seamless_no_z_x` coordinates must be of `vector3`
- `calc_seamless_no_z_y` coordinates must be of `vector3`
- `calc_seamless_no_z_z` coordinates must be of `vector4`
- `calc_seamless_no_z_xy` coordinates must be of `vector4`
- `calc_seamless_no_z_xz` coordinates must be of `vector8`
- `calc_seamless_no_z_yz` coordinates must be of `vector8`
- `calc_seamless_no_z_xyz` coordinates must be of `vector8`

### 2.4.4 Scale To Range

```
REAL* scaleToRange(REAL *data, size_t
    count, REAL min, REAL max, REAL
    low, REAL high);
```

Listing 9: Definition of scaleToRange function

```
scaleToRange(data, count, min, max, 0,
    1);
```

Listing 10: Example for scaleToRange function. The data will be scaled to fit between 0 and 1.

Defined in `imaging.h`. Scales the values in data to the range between low

and high. The data is modified by this function.

## 2.5 Noise Generation Functions

Since *ANL-OpenCL* is a port of the Josua Tippetts' C++ library http://accidentalnoise.sourceforge.net/index.html the same documentation can be used for the noise generation functions. All of the functions and types are defined in the `noise_gen.h`.

### 2.5.1 Value Noise Functions

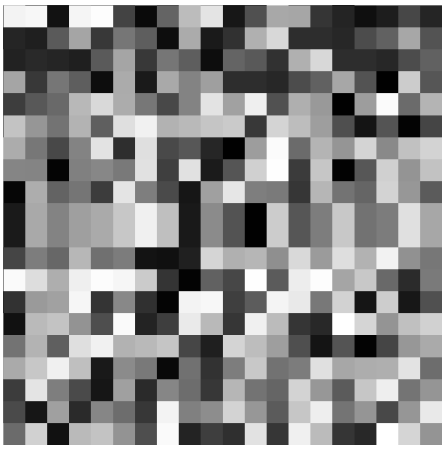


Figure 9: Value Noise 3D no interpolation.

```
REAL value_noise2D(vector2 v, uint
    seed, interp_func interp);
REAL value_noise3D(vector3 v, uint
    seed, interp_func interp);
REAL value_noise4D(vector4 v, uint
    seed, interp_func interp);
REAL value_noise6D(vector8 v, uint
    seed, interp_func interp);
```

Listing 11: Definition of value noise functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
```

Figure 10: Value Noise 3D linear interpolation.

Figure 11: Value Noise 3D hermite interpolation.

```
    const float v = value_noise3D(
    coord[i], seed, linearInterp);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 12: Example for value noise functions

Value noise functions.

> Value noise is a type of noise commonly used as a procedural texture primitive in computer graphics. This method consists of the creation of a lattice of points which are assigned random values. The

Figure 12: Value Noise 3D quintic interpolation.

noise function then returns the interpolated number based on the values of the surrounding lattice points. https://en.wikipedia.org/wiki/Value_noise

### 2.5.2 Gradient Noise Functions

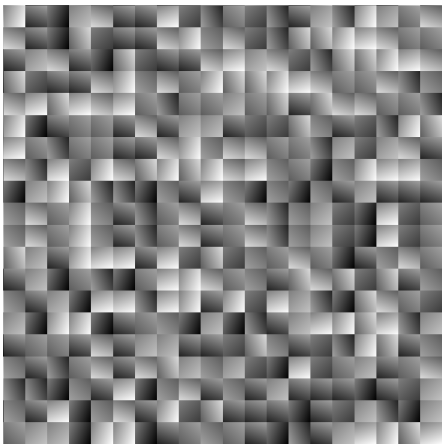


Figure 13: Gradient Noise 3D no interpolation.

```
REAL gradient_noise2D(vector2 v, uint seed, interp_func interp);
REAL gradient_noise3D(vector3 v, uint seed, interp_func interp);
REAL gradient_noise4D(vector4 v, uint seed, interp_func interp);
REAL gradient_noise6D(vector8 v, uint seed, interp_func interp);
```

Listing 13: Definition of gradient noise functions

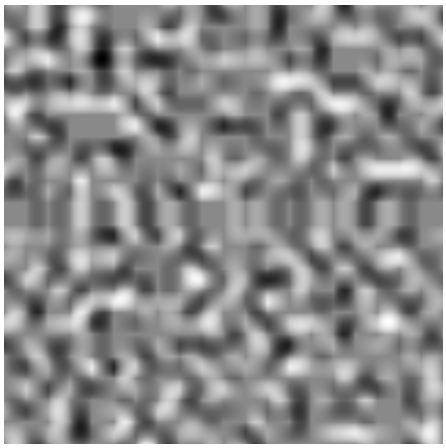


Figure 14: Gradient Noise 3D linear interpolation.

Figure 15: Gradient Noise 3D hermite interpolation.

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    const float v = gradient_noise3D(coord[i], seed, linearInterp);
    write_imagef(output, (int2)(g0, g1), (float4)(v, v, v, 1.0));
}
```

Listing 14: Example for gradient noise functions

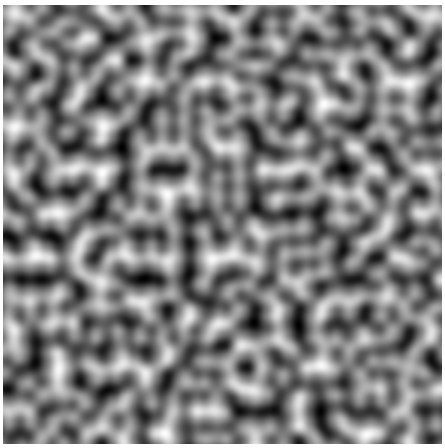


Figure 16: Gradient Noise 3D quintic interpolation.

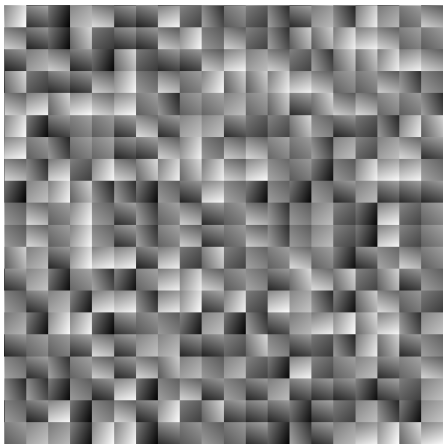


Figure 17: Gradval Noise 3D no interpolation.

Gradient noise functions.

> Gradient noise is a type of noise commonly used as a procedural texture primitive in computer graphics. This method consists of a creation of a lattice of random (or typically pseudorandom) gradients, dot products of which are then interpolated to obtain values in between the lattices. An artifact of some implementations of this noise is that the returned value at the lattice points is 0. Unlike the value noise, gradient noise has more energy in the high frequencies. https://en.wikipedia.org/wiki/Gradient_noise

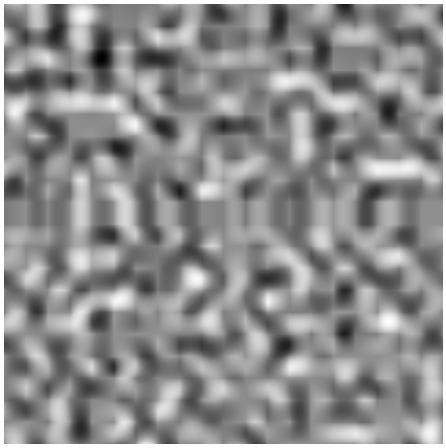


Figure 18: Gradval Noise 3D linear interpolation.

### 2.5.3 Gradval Noise Functions

```
REAL gradval_noise2D(vector2 v, uint seed, interp_func interp);
REAL gradval_noise3D(vector3 v, uint seed, interp_func interp);
REAL gradval_noise4D(vector4 v, uint seed, interp_func interp);
REAL gradval_noise6D(vector8 v, uint seed, interp_func interp);
```

Listing 15: Definition of gradval noise functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    const float v = gradval_noise3D(coord[i], seed, linearInterp);
    write_imagef(output, (int2)(g0, g1), (float4)(v, v, v, 1.0));
}
```

Listing 16: Example for gradval noise functions

Combined value and gradient noise

Figure 19: Gradval Noise 3D hermite interpolation.

Figure 20: Gradval Noise 3D quintic interpolation.

functions.

### 2.5.4 White Noise Functions

```
REAL white_noise2D(vector2 v, uint
    seed, interp_func interp);
REAL white_noise3D(vector3 v, uint
    seed, interp_func interp);
REAL white_noise4D(vector4 v, uint
    seed, interp_func interp);
REAL white_noise6D(vector8 v, uint
    seed, interp_func interp);
```

Listing 17: Definition of white noise functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
```

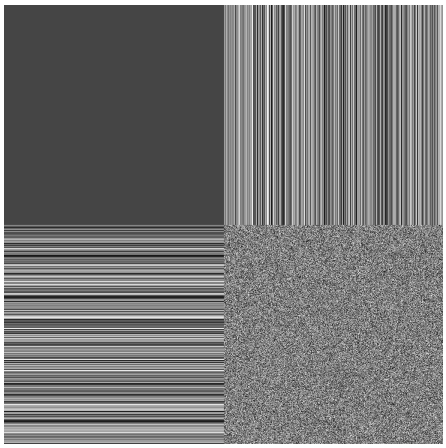


Figure 21: White Noise 3D no interpolation.

```
) {
    $insert_localMapRange
    long seed = 200;
    const float v = white_noise3D(
    coord[i], seed, linearInterp);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 18: Example for white noise functions

White noise functions. The interpolation function parameter is not used. The interpolation function parameter is only for compatibility with the other noise functions.

> In signal processing, white noise is a random signal having equal intensity at different frequencies, giving it a constant power spectral density. https://en.wikipedia.org/wiki/White_noise

### 2.5.5 Simplex Noise Functions

```
REAL simplex_noise2D(vector2 v, uint
    seed, interp_func interp);
REAL simplex_noise3D(vector3 v, uint
    seed, interp_func interp);
REAL simplex_noise4D(vector4 v, uint
    seed, interp_func interp);
```

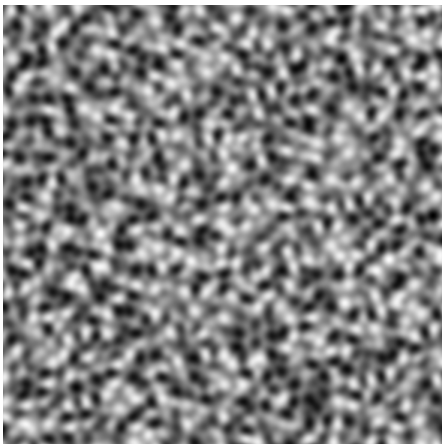


Figure 22: Simplex Noise 3D no interpolation.

```
REAL simplex_noise6D(vector8 v, uint
    seed, interp_func interp);
```

Listing 19: Definition of simplex noise functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    const float v = simplex_noise3D(
    coord[i], seed, linearInterp);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 20: Example for simplex noise functions

Simplex noise functions. The interpolation function parameter is not used. The interpolation function parameter is only for compatibility with the other noise functions.

> Simplex noise is a method for constructing an n-dimensional noise function comparable to Perlin noise ("classic" noise) but with fewer directional artifacts and, in higher dimensions, a lower computational overhead. https://en.wikipedia.org/wiki/Simplex_noise

### 2.5.6 Cellular Noise Functions

```
REAL cellular_function2D(vector2 v,
    uint seed, REAL *f, REAL *disp,
    dist_func2 distance);
REAL cellular_function3D(vector3 v,
    uint seed, REAL *f, REAL *disp,
    dist_func3 distance);
REAL cellular_function4D(vector4 v,
    uint seed, REAL *f, REAL *disp,
    dist_func4 distance);
REAL cellular_function6D(vector8 v,
    uint seed, REAL *f, REAL *disp,
    dist_func6 distance);
```

Listing 21: Definition of cellular noise functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    REAL f[4] = { 10, 5, 2.5, 1.25 };
    REAL disp[4] = { 100, 50, 25, 10
    };
    const float v =
    cellular_function3D(coord[i], seed
    , f, disp, distEuclid3);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 22: Example for cellular noise functions

Cellular noise functions. Compute distance (for cellular modules) and displacement (for voronoi modules).

### 2.5.7 Interpolation Functions

```
REAL noInterp(REAL t);
REAL linearInterp(REAL t);
REAL hermiteInterp(REAL t);
REAL quinticInterp(REAL t);
```

Listing 23: Definition of interpolation functions

```
const float v = cellular_noise3D(coord
    [i], seed, noInterp);
// or
const float v = cellular_noise3D(coord
    [i], seed, linearInterp);
// or
const float v = cellular_noise3D(coord
    [i], seed, hermiteInterp);
// or
```

```
const float v = cellular_noise3D(coord
    [i], seed, quinticInterp);
```

Listing 24: Example for interpolation functions

The four interpolation functions are provided for the noise generation functions. The functions are not used directly but specified as a parameter.

```
typedef REAL (*interp_func)(REAL);
```

Listing 25: Definition of interpolation function type

The interpolation functions take an input of the type `REAL` that is either type `float` or `double` and returns the interpolated value that is also of type `REAL`.

### 2.5.8 Noise Generation Function Types

```
typedef REAL (*noise_func2)(vector2,
    uint, interp_func);
typedef REAL (*noise_func3)(vector3,
    uint, interp_func);
typedef REAL (*noise_func4)(vector4,
    uint, interp_func);
typedef REAL (*noise_func6)(vector8,
    uint, interp_func);
```

Listing 26: Definition of noise generation function types

The noise generation functions can be used as parameters to other generation functions. Those are the definitions for 2D, 3D, 4D and 6D noise functions types. The parameters are described below.

1. expects the 2D, 3D, 4D or 6D coordinate;
2. expects the seed;
3. expects interpolation function;

### 2.5.9 Distance Functions

```
REAL distEuclid2(vector2 a, vector2 b)
    ;
REAL distEuclid3(vector3 a, vector3 b)
    ;
REAL distEuclid4(vector4 a, vector4 b)
    ;
REAL distEuclid6(vector8 a, vector8 b)
    ;
```

Listing 27: Definition of Euclid distance functions

Returns the Euclidean distance between two points $a$ and $b$ as $d(a,b) = |a-b|$.

```
REAL distManhattan2(vector2 a, vector2
    b);
REAL distManhattan3(vector3 a, vector3
    b);
REAL distManhattan4(vector4 a, vector4
    b);
REAL distManhattan6(vector8 a, vector8
    b);
```

Listing 28: Definition of Manhattan distance functions

Returns the Manhattan distance between two points.

> A taxicab geometry is a form of geometry in which the usual distance function or metric of Euclidean geometry is replaced by a new metric in which the distance between two points is the sum of the absolute differences of their Cartesian coordinates. https://en.wikipedia.org/wiki/Taxicab_geometry

```
REAL distGreatestAxis2(vector2 a,
    vector2 b);
REAL distGreatestAxis3(vector3 a,
    vector3 b);
REAL distGreatestAxis4(vector4 a,
    vector4 b);
REAL distGreatestAxis6(vector8 a,
    vector8 b);
```

Listing 29: Definition of greatest axis distance functions

Returns the distance between two points on the axis that have the greatest distance.

```
REAL distLeastAxis2(vector2 a, vector2
    b);
REAL distLeastAxis3(vector3 a, vector3
    b);
```

```
REAL distLeastAxis4(vector4 a, vector4
    b);
REAL distLeastAxis6(vector8 a, vector8
    b);
```

Listing 30: Definition of least axis distance functions

Returns the distance between two points on the axis that have the least distance.

### 2.5.10 Distance Function Types

```
typedef REAL (*dist_func2)(vector2,
    vector2);
typedef REAL (*dist_func3)(vector3,
    vector3);
typedef REAL (*dist_func4)(vector4,
    vector4);
typedef REAL (*dist_func6)(vector8,
    vector8);
```

Listing 31: Definition of distance function types

The distance functions can be used as parameters to other generation functions. Those are the definitions for 2D, 3D, 4D and 6D noise functions types. The parameters are described below. Returns the distance between the two coordinates.

1. expects the first 2D, 3D, 4D or 6D coordinate;

2. expects the second 2D, 3D, 4D or 6D coordinate;

## 2.6 Kernel Functions

Since *ANL-OpenCL* is a port of the Josua Tippetts' C++ library http://accidentalnoise.sourceforge.net/index.html the same documentation can be used for the noise generation functions. All of the functions and types are defined in the `kernel.h`.

### 2.6.1 Manipulation Functions

```
vector3 rotateDomain3(vector3 src,
    REAL angle, REAL ax, REAL ay, REAL
     az);
vector4 rotateDomain4(vector4 src,
    REAL angle, REAL ax, REAL ay, REAL
     az);
vector8 rotateDomain6(vector8 src,
    REAL angle, REAL ax, REAL ay, REAL
     az);
```

Listing 32: Definition of rotate domain functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    vector3 rotated = rotateDomain3(
    coord[i], radians(90), 1, 0, 0);
    const float v = value_noise3D(
    rotated, seed, linearInterp);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 33: Example for rotate domain functions

Rotates the source coordinates by the angle over the rotation axis.

```
scaleDomain(v, scale)
```

Listing 34: Definition of scale domain function

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    vector3 scaled = scaleDomain(coord
    [i], 10);
    const float v = value_noise3D(
    scaled, seed, linearInterp);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 35: Example for scale domain function

Multiplies the source coordinates by the scale.

```
vector4 combineRGBA(REAL r, REAL g,
    REAL b, REAL a);
vector4 combineHSVA(REAL h, REAL s,
    REAL v, REAL a);
```

Listing 36: Definition of combine color values functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    const float r = value_noise3D(
    coord[i], seed, linearInterp);
    const float g = value_noise3D(
    coord[i], seed*10, linearInterp);
    const float b = value_noise3D(
    coord[i], seed*100, linearInterp);
    const vector4 c = combineRGBA(r, g
    , b, 1.0);
    write_imagef(output, (int2)(g0, g1
    ), c);
}
```

Listing 37: Example for scale domain function

Combines the RGBA or HSVA values.

### 2.6.2 Fractal Layer Functions

Figure 23: Simple fractal layer with value noise 3D and hermite interpolation with rotation.

```
REAL simpleFractalLayer3(vector3 v,
    noise_func3 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
```

Figure 24: Simple fractal layer with gradient noise 3D and linear hermite with rotation.

Figure 25: Simple fractal layer with gradval noise 3D and hermite interpolation with rotation.

```
REAL simpleFractalLayer4(vector4 v,
    noise_func4 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
REAL simpleFractalLayer6(vector8 v,
    noise_func6 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
```

Listing 38: Definition of simple fractal layer functions

```
kernel void map2d_image(
```

Figure 26: Simple fractal layer with simplex noise 3D with rotation.

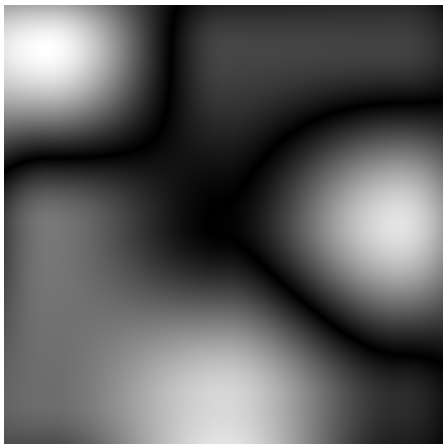


Figure 27: Ridged fractal layer with value noise 3D and hermite interpolation with rotation.

```
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    // no rotation
    const float v =
    simpleFractalLayer3(coord[i],
    value_noise3D, 200, noInterp, 1,
    0.125, false, 0.0, 0.0, 0.0, 0.0);
    // with rotation
    const float v =
    simpleFractalLayer3(coord[i],
    value_noise3D, 200, noInterp, 1,
    0.125, true, 1.57, 1.0, 0.0, 0.0);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 39: Example for simple fractal layer functions

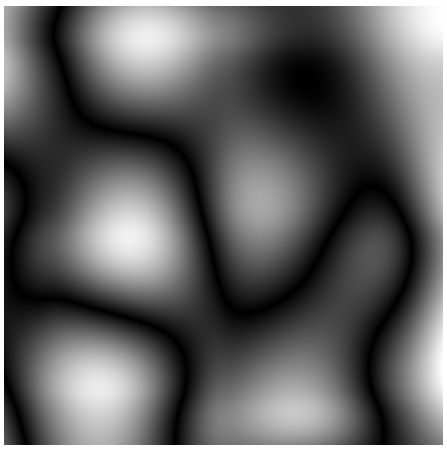


Figure 28: Ridged fractal layer with gradient noise 3D and linear hermite with rotation.

Returns simple fractal layer value for the coordinate.

```
REAL simpleRidgedLayer3(vector3 v,
    noise_func3 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
REAL simpleRidgedLayer4(vector4 v,
    noise_func4 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
REAL simpleRidgedLayer6(vector8 v,
    noise_func6 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
```

Listing 40: Definition of ridged fractal layer functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    // no rotation
    const float v = simpleRidgedLayer3
    (coord[i], value_noise3D, 200,
    noInterp, 1, 0.125, false, 0.0,
```

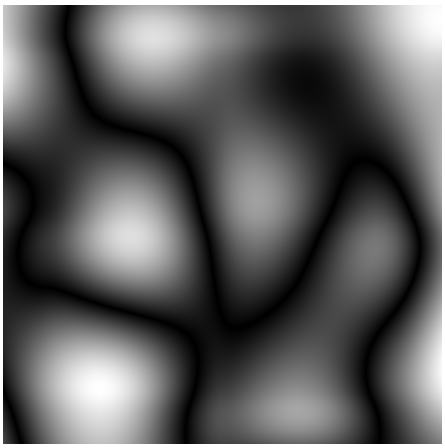


Figure 29: Ridged fractal layer with gradval noise 3D and hermite interpolation with rotation.

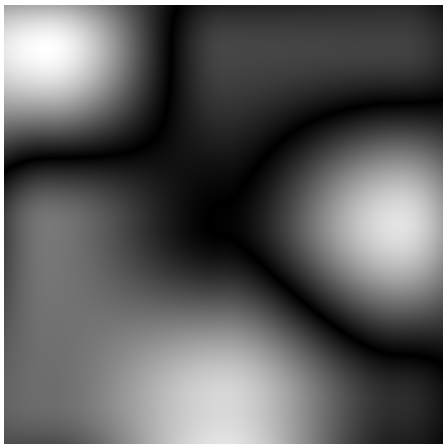


Figure 31: Billow fractal layer with value noise 3D and hermite interpolation with rotation.

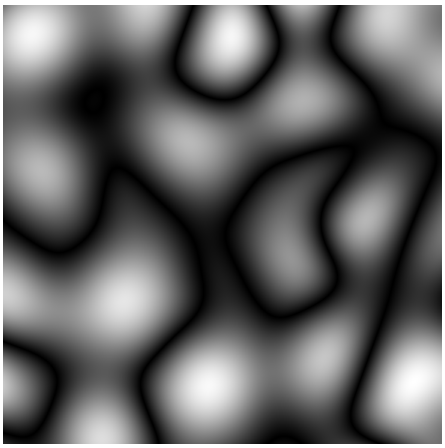


Figure 30: Ridged fractal layer with simplex noise 3D with rotation.

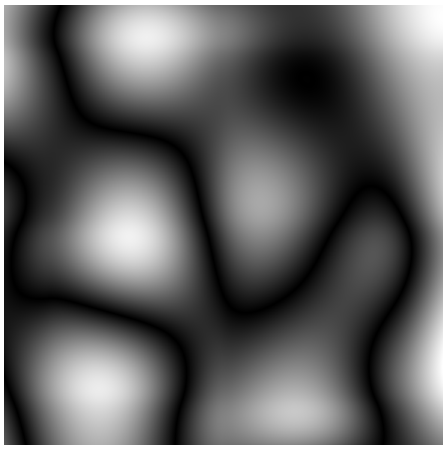


Figure 32: Billow fractal layer with gradient noise 3D and linear hermite with rotation.

```
    0.0, 0.0, 0.0);
    // with rotation
    const float v = simpleRidgedLayer3
    (coord[i], value_noise3D, 200,
    noInterp, 1, 0.125, true, 1.57,
    1.0, 0.0, 0.0);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 41: Example for ridged fractal layer functions

Returns simple ridged layer value for the coordinate.

```
REAL simpleBillowLayer3(vector3 v,
    noise_func3 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
REAL simpleBillowLayer4(vector4 v,
    noise_func4 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
     REAL az);
REAL simpleBillowLayer6(vector8 v,
    noise_func6 basistype, uint seed,
    interp_func interp, REAL
    layerscale, REAL layerfreq, bool
    rot, REAL angle, REAL ax, REAL ay,
```

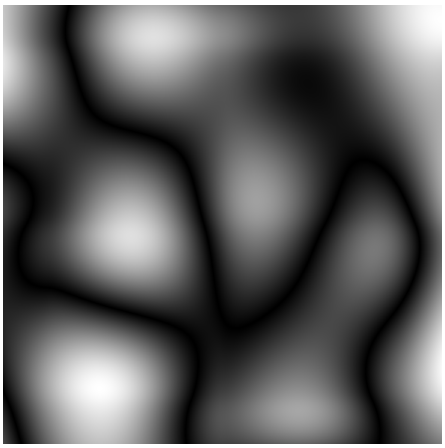


Figure 33: Billow fractal layer with gradval noise 3D and hermite interpolation with rotation.

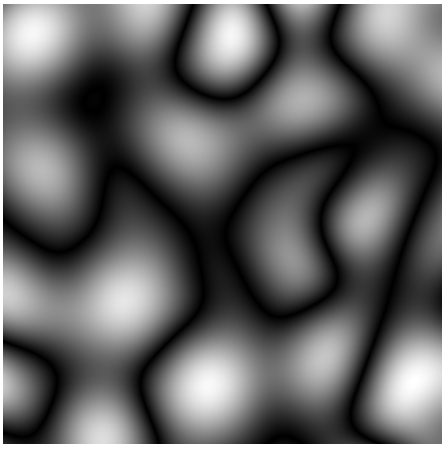


Figure 34: Billow fractal layer with simplex noise 3D with rotation.

```
    REAL az);
```

Listing 42: Definition of billow fractal layer functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    // no rotation
    const float v = simpleBillowLayer3
    (coord[i], value_noise3D, 200,
    noInterp, 1, 0.125, false, 0.0,
    0.0, 0.0, 0.0);
    // with rotation
    const float v = simpleBillowLayer3
    (coord[i], value_noise3D, 200,
    noInterp, 1, 0.125, true, 1.57,
    1.0, 0.0, 0.0);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 43: Example for billow fractal layer functions

Returns simple billow layer value for the coordinate.

### 2.6.3 Fractal Functions

Figure 35: Simple brownian motion fractal with value noise 3D and hermite interpolation with rotation.

```
simplefBm3(vector3 v, noise_func3
    basistype, uint seed, interp_func
    interp, random_func rnd, void *
    srnd, uint numoctaves, REAL
    frequency, bool rot);
simplefBm4(vector4 v, noise_func4
    basistype, uint seed, interp_func
    interp, random_func rnd, void *
    srnd, uint numoctaves, REAL
    frequency, bool rot);
simplefBm6(vector8 v, noise_func6
    basistype, uint seed, interp_func
    interp, random_func rnd, void *
    srnd, uint numoctaves, REAL
    frequency, bool rot);
```

Listing 44: Definition of simple brownian motion fractal functions

Figure 36: Simple brownian motion fractal with gradient noise 3D and linear hermite with rotation.

Figure 37: Simple brownian motion fractal with gradval noise 3D and hermite interpolation with rotation.

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    kiss09_state srnd;
    kiss09_seed(&srnd, 200);
    // no rotation
    const float v = simplefBm3(coord[i
    ], value_noise3D, 200, noInterp,
    random_kiss09, &srnd, 3, 0.125,
    false);
    // with rotation
    const float v = simplefBm3(coord[i
```

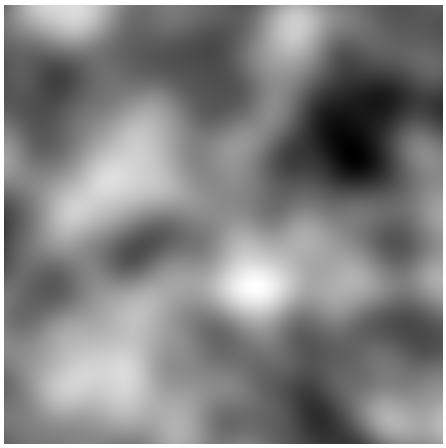


Figure 38: Simple brownian motion fractal with simplex noise 3D with rotation.

```
    ], value_noise3D, 200, noInterp,
    random_kiss09, &srnd, 3, 0.125,
    true);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 45: Example for simple brownian motion fractal functions

Returns fractional brownian motion value for the coordinate.

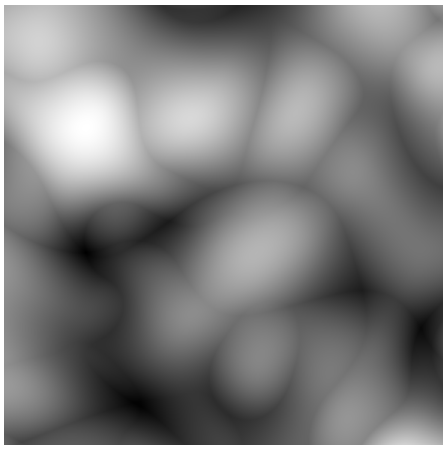


Figure 39: Simple ridged-multifractal with value noise 3D and hermite interpolation with rotation.

```
simpleRidgedMultifractal3(vector3 v,
    noise_func3 basistype, uint seed,
```

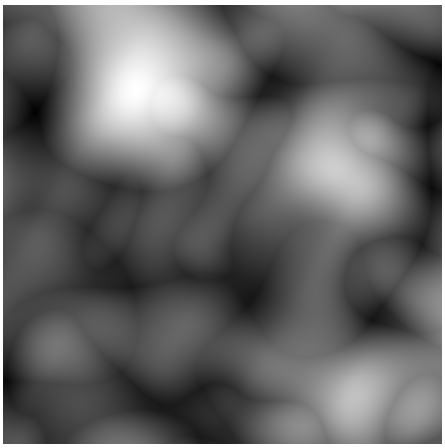


Figure 40: Simple ridged-multifractal with gradient noise 3D and linear hermite with rotation.

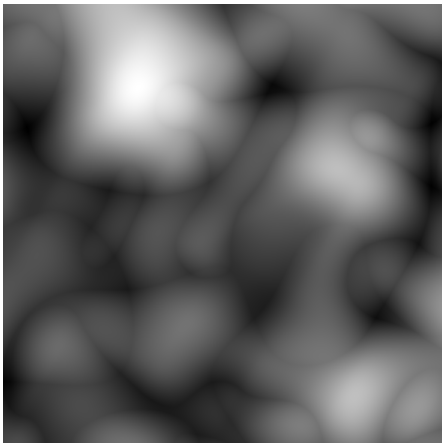


Figure 41: Simple ridged-multifractal with gradval noise 3D and hermite interpolation with rotation.

```
    interp_func interp, random_func
    rnd, void *srnd, uint numoctaves,
    REAL frequency, bool rot);
simpleRidgedMultifractal4(vector4 v,
    noise_func4 basistype, uint seed,
    interp_func interp, random_func
    rnd, void *srnd, uint numoctaves,
    REAL frequency, bool rot);
simpleRidgedMultifractal6(vector8 v,
    noise_func6 basistype, uint seed,
    interp_func interp, random_func
    rnd, void *srnd, uint numoctaves,
```

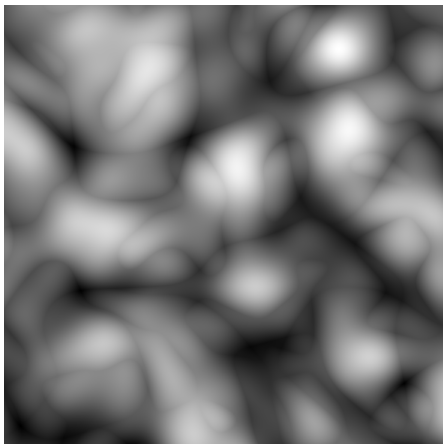


Figure 42: Simple ridged-multifractal with simplex noise 3D with rotation.

```
    REAL frequency, bool rot);
```

Listing 46: Definition of simple brownian motion fractal functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    kiss09_state srnd;
    kiss09_seed(&srnd, 200);
    // no rotation
    const float v =
    simpleRidgedMultifractal3(coord[i
    ], value_noise3D, 200, noInterp,
    random_kiss09, &srnd, 3, 0.125,
    false);
    // with rotation
    const float v =
    simpleRidgedMultifractal3(coord[i
    ], value_noise3D, 200, noInterp,
    random_kiss09, &srnd, 3, 0.125,
    true);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 47: Example for simple brownian motion fractal functions

Returns ridged-multifractal noise value for the coordinate.

```
simpleBillow3(vector3 v, noise_func3
    basistype, uint seed, interp_func
```

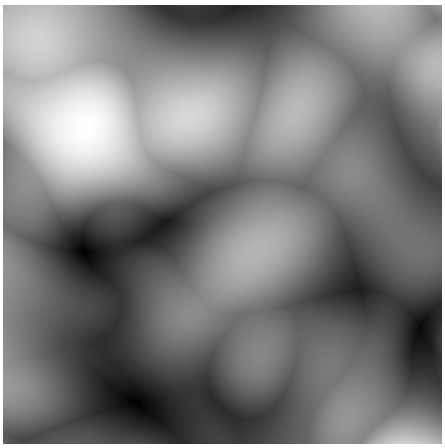


Figure 43: Simple billow fractal with value noise 3D and hermite interpolation with rotation.

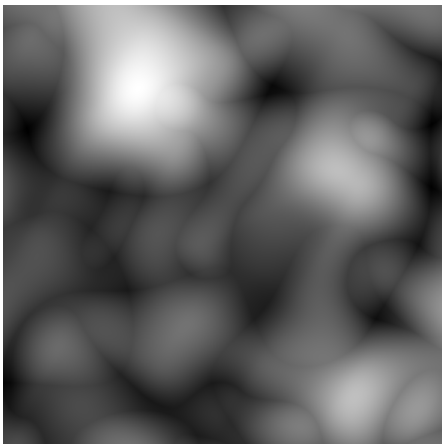


Figure 45: Simple billow fractal with gradval noise 3D and hermite interpolation with rotation.

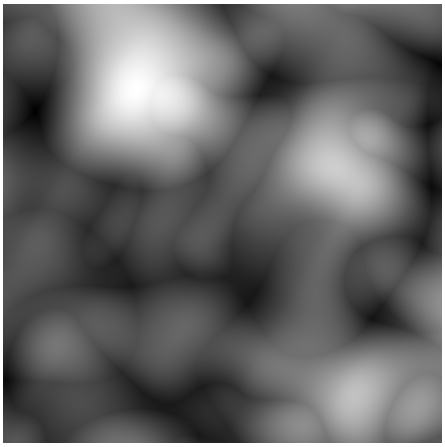


Figure 44: Simple billow fractal with gradient noise 3D and linear hermite with rotation.

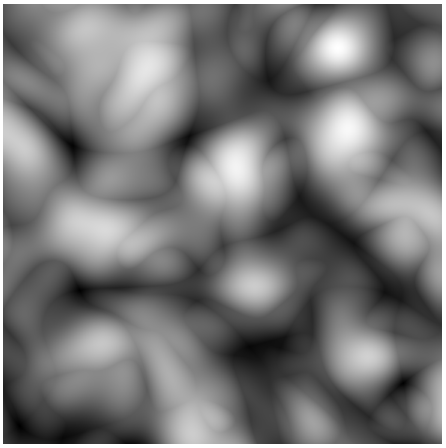


Figure 46: Simple billow fractal with simplex noise 3D with rotation.

```
    interp, random_func rnd, void *
    srnd, uint numoctaves, REAL
    frequency, bool rot);
simpleBillow4(vector4 v, noise_func4
    basistype, uint seed, interp_func
    interp, random_func rnd, void *
    srnd, uint numoctaves, REAL
    frequency, bool rot);
simpleBillow6(vector8 v, noise_func6
    basistype, uint seed, interp_func
    interp, random_func rnd, void *
    srnd, uint numoctaves, REAL
    frequency, bool rot);
```

Listing 48: Definition of simple billow fractal functions

```
kernel void map2d_image(
```

```
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    long seed = 200;
    kiss09_state srnd;
    kiss09_seed(&srnd, 200);
    // no rotation
    const float v = simpleBillow3(
    coord[i], value_noise3D, 200,
    noInterp, random_kiss09, &srnd, 3,
     0.125, false);
    // with rotation
    const float v = simpleBillow3(
    coord[i], value_noise3D, 200,
    noInterp, random_kiss09, &srnd, 3,
```

```
     0.125, true);
    write_imagef(output, (int2)(g0, g1
    ), (float4)(v, v, v, 1.0));
}
```

Listing 49: Example for simple billow fractal functions

Returns billow (cloud-like, lumpy) fractal value for the coordinate.

### 2.6.4 Other Functions

```
x(v)
y(v)
z(v)
w(v)
u(v)
v(v)
```

Listing 50: Definition of components functions

```
kernel void map2d_image(
global struct SMappingRanges *g_ranges
    ,
write_only image2d_t output
) {
    $insert_localMapRange
    const float x = x(coord[i]);
    const float y = y(coord[i]);
    const float z = z(coord[i]);
    const float w = w(coord[i]);
    const float u = u(coord[i]);
    const float v = v(coord[i]);
}
```

Listing 51: Example for components functions

Returns the x, y, z, w, u and v component of the vector.

```
typedef REAL (*random_func)(void*);
```

Listing 52: Definition of random function type

Function that returns a random number.

## Index