FracNoise

P. Baillehache

June 10, 2018

Contents

Introduction

FracNoise is a C library providing structures and functions to generate noise based on the Perlin noise.

The library provides the structure PerlinNoise and its functions to generate noise as described by Prof. Ken Perlin.

It also provides the structure FracNoise and its functions, which extends the PerlinNoise. A FracNoise is made of a composition of one or several PerlinNoise. Each of the composed PerlinNoise can be individually controled over its input/output values scale and shift; be attached to a boundary (Shapoid) outside of which the PerlinNoise is inactive, and inside of which it becomes smoothly active across a border width defined by the user; support a recursive definition with user defined depth and strength; have a 'smooth' parameter to set the aspect (round or spiky) of the PerlinNoise; have a user defined seed; return output as a vector of any dimension; take input as a vector of 1, 2 or 3 dimensions, however the limit of 3 can somehow be extended up to 6 by using the 3 components of the seed as input values too.

The FracNoise structure also provides an export function toward ".df3" density file to be used in computer graphics software like POV-Ray.

It uses the `PBErr`, the `GSet`, the `Shapoid` and the `PBMath` library.

1 Definitions

1.1 Perlin noise

For details about the Perlin noise, please refer to the paper of, and its reference implementation by, Prof. Ken Perlin:
http://mrl.nyu.edu/~perlin/paper445.pdf
http://mrl.nyu.edu/~perlin/noise/

The implementation of the Perlin noise in the FracNoise library is a port from the JAVA Reference Implementation of Prof. Perlin toward C language with only modification a mapping of the output value to [0.0, 1.0].

1.2 FracNoise

In the FracNoise structure, each instance of a PerlinNoise is called a PerlinNoisePod. A PerlinNoisePod contains a PerlinNoise and the associated

parameters. These parameters are described below:

- seed: a 3D vector of float values.
  The value returned by the PerlinNoise is defined by the permutations used to init the structure. Then a given input will always generates the same output, which is not very useful. Different output values would require to generate different permutations. FracNoise offers another solution: Each PerlinNoise is associated to a seed which is used to transform the input values. The transformation is a rotation of the input vector around the axis defined by the seed vector. The seed vector is a 3D vector, and the input is extended internally to a 3D vector if necessary (missing components equal 0.0). The angle of the rotation for the i-th component of the output value is defined as $2\pi(i+1)/(n+1)$ where n is the dimension of output. Then, we can generate as many different noises as needed, and we solve at the same time the problem of generating output of dimension higher than 1. Also, the seed components can be used as extra components to the input when one needs more than 3D input. As the transformation is a 3D rotation of the input, continuity and derivability of the output are conserved over seed values.

- scaleIn: a vector of float values of dimension equals to the dimension of the input.
  This scale is applied to the input, component by component.

- fractalLvl: an integer greater than or equal to 0.
  The fractal level controls the number of recursive call of the PerlinNoise on itself. If it's equal to 0, the output is the standard output. If it's equal to 1 the output is equal to the standard output added to the output of the same PerlinNoise multiplied by the fractal coefficient (see below) for input divided by the fractal coefficient. And so on (there is no limit to the fractal level). Recursive call are made before applying the scaling and shifting on output. As the output at each level sum up, the final output value might get out of the range [0.0, 1.0], but it is automatically rescaled to ensure the values stay in the correct range whatever the fractal level. Recursive call occurs before smoothing, scaling and shifting.

- fractalCoeff: a float value (typically in]0.0, 1.0[but one may experiments with values out of this range).
  The output value is multiplied by the fractal coefficient at each fractal

level, and the input value is divided by the fractal coefficient at each level. the output value is rescaled to ensure it stays inside [0.0,1.0]. So for example, for a PerlinNoise with a fractal level of 2, a fractal coefficient of 0.1, an input value of 0.5, the output value will be equal to (PerlinNoise(0.5)*0.9+0.1*PerlinNoise(5.0))*0.9+0.01*PerlinNoise(50.0)

- smooth: a float value greater than 0.0.
  The smooth is applied after fractal and before scaling.Output values are raise to the power of smooth component by component. This can be used to control the general aspect of the distribution of the output values (more bumpy for value below 1.0, more thorny for value above).

- square: a float value greater in [0.0, 1.0].
  The squareness of the pod controls the smoothing of the input value of the PerlinNoise. If it equals 0.0 the result is the standard PerlinNoise. If it equals 1.0 the result is the PerlinNoise without smoothing of the input. Explicitly: $smooth(t, square) = (1-square)smooth(t)+square*t$. The squareness is applied when calculating the PerlinNoise for a pod with this pod's squareness.

- bound: a Shapoid.
  The PerlinNoise is defined over $\mathbb{R}$, however one can want to restrain it to some subdomain. This can be done by attaching a Shapoid with proper position and axis of dimension equal to the input dimension. Then, the PerlinNoise will output 0.0 for input (original one, not the scaleIn-ed one) outside of the subdomain defined by the outside of the Shapoid, and its normal value for the subdomain defined by the inside of the Shapoid.

- border: a float value between 0.0 and 1.0 representing the border of the boundary.
  If the PerlinNoise is attached to a boundary, one may want a smooth transition between the outside and inside of the boundary. The depth in the Shapoid defining the boundary of the input position is compared to the border, if the depth is greater than the border then the normal output is returned. If the depth is between 0.0 and the border, the normal output is multiplied by SmootherStep(depth/border).

- scaleOut: a vector of float values of dimension equals to the dimension of the output.
  This scale is applied to the output, component by component.

- shiftOut: a vector of float values of dimension equals to the dimension of the output.
  This shift is added to the output after scaling.

The value of a FracNoise for a given input is equal to the sum of the values of its PerlinNoisePod for this input and can be expressed as follow:

$$FracNoise(\overrightarrow{p}) = \sum_{pod}\left(\left(In(\overrightarrow{p}).\left(\sum_{f=0}^{l} c^f\left[PN\left(Rot_{\overrightarrow{e}}\left(\overrightarrow{p}\odot\overrightarrow{si},\frac{2\pi(i+1)}{n+1}\right).\frac{1.0}{c^f},q\right)^s\right]_{i\in[0,n-1]}\odot\overrightarrow{so}+\overrightarrow{t}\right)\right)\right) \quad (1)$$

where

- $\overrightarrow{p}$ is the input (redimensioned to a 3D vector if it's of different dimension).
- *pod* is the set of PerlinNoisePod.
- $In(\overrightarrow{p})$ is the insideness of $\overrightarrow{p}$, equals to 1.0 if the PerlinNoisePod has no boundary, or if it has a boundary: 0.0 if $\overrightarrow{p}$ is out of boundary; 1.0 if $depth(\overrightarrow{p})$ is greater than the *border* of the PerlinNoisePod; SmootherStep($depth(\overrightarrow{p})/border$) else. We remind that SmootherStep(x)=$((6x-15)x+10)x^3$.
- l is the fractalLvl.
- c is the fractalCoeff.
- $PN()$ is the PerlinNoise function.
- $\overrightarrow{e}$ is the seed.
- $Rot_{\overrightarrow{e}}(\overrightarrow{p},\theta)$ is the rotation of $\overrightarrow{p}$ around the axis $\overrightarrow{e}$ by angle θ.
- $\overrightarrow{si}$ is the scaleIn.
- $\overrightarrow{so}$ is the scaleOut.
- $\overrightarrow{t}$ is the shiftOut.
- n is the number of dimensions of output.
- s is the smoothness of the pod.
- q is the squareness of the pod.

2 Interface

```
// ============ FRACNOISE.H ================

#ifndef FRACNOISE_H
#define FRACNOISE_H

// ================= Include =================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pbmath.h"
#include "gset.h"
#include "shapoid.h"

// -------------- PerlinNoise
// http://mrl.nyu.edu/~perlin/noise/

// ================= Data structure ==================

typedef struct PerlinNoise {
  // Parameters
  int _p[512];
} PerlinNoise;

// ================ Functions declaration ====================

// Return a new PerlinNoise
PerlinNoise* PerlinNoiseCreate(void);

// Return a new static PerlinNoise
PerlinNoise PerlinNoiseCreateStatic(void);

// Free the memory used by the PerlinNoise 'that'
void PerlinNoiseFree(PerlinNoise** that);

// Return the value of the PerlinNoise 'that' at position 'u'
// 'u' can be of dimension 1 to 3, 'u' values in R, the Perlin noise
// has interpolation on unit, so u in [0.0, 1.0] will be interpolated
// between 2 values, u in [0.0, 2.0] will be interpolated between
// 3 values, and the noise always equal 0.5 at unit values, so
// noise(0.0) == noise(1.0) == noise(2.0) == ... == 0.5
// _PerlinNoiseGet3D takes a third argument which is the squareness
// in [0.0,1.0]
// Return a value in [0.0, 1.0]
float _PerlinNoiseGet(const PerlinNoise* const that,
  const VecFloat* const u, const float squareness);
float _PerlinNoiseGet1D(const PerlinNoise* const that, const float u,
  const float squareness);
float _PerlinNoiseGet2D(const PerlinNoise* const that,
  const VecFloat2D* const u, const float squareness);
float _PerlinNoiseGet3D(const PerlinNoise* const that,
  const VecFloat3D* const u, const float squareness);

// Set the permutations of the PerlinNoise 'that' to 'permut'
// 'permut' is an array of 256 int
void PerlinNoiseSetPermut(PerlinNoise* const that,
  const int* const permut);
```

```
// -------------- PerlinNoisePod

typedef struct PerlinNoisePod {
  // Perlin noise
  PerlinNoise _noise;
  // Seed
  VecFloat3D _seed;
  // Boundary
  Shapoid* _bound;
  // Boundary border
  float _border;
  // Scale inputs
  VecFloat* _scaleIn;
  // Scale outputs
  VecFloat* _scaleOut;
  // Shift outputs
  VecFloat* _shiftOut;
  // Fractal level
  int _fractalLvl;
  // Fractal coefficient
  float _fractalCoeff;
  // Smoothness
  float _smooth;
  // Squareness
  float _square;
} PerlinNoisePod;

// ================ Functions declaration ====================

// Return a new PerlinNoisePod with dimension 'dim' and seed
// 'seed'
// If the seed is the vector null replace it by (1,1,1)
// Default values:
// bound = NULL, border = 0.1, scaleIn/Out = 1.0
// fractalLvl = 0, fractalCoeff = 0.1, smooth = 1.0
PerlinNoisePod* PerlinNoisePodCreate(const VecShort2D* const dim,
  const VecFloat3D* const seed);

// Free memory used by the PerlinNoisePod 'that'
void PerlinNoisePodFree(PerlinNoisePod** that);

// Get the Perlin noise of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
const PerlinNoise* PerlinNoisePodNoise(const PerlinNoisePod* const that);

// Get the seed of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat3D* PerlinNoisePodSeed(const PerlinNoisePod* const that);

// Get the bound of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
Shapoid* PerlinNoisePodBound(const PerlinNoisePod* const that);

// Get the scale of inputs of the PerlinNoisePod 'that'
#if BUILDMODE != 0
```

```
inline
#endif
VecFloat* PerlinNoisePodScaleIn(const PerlinNoisePod* const that);

// Get the scale of outputs of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* PerlinNoisePodScaleOut(const PerlinNoisePod* const that);

// Get the shift of outputs of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* PerlinNoisePodShiftOut(const PerlinNoisePod* const that);

// Get the fractal level of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
int PerlinNoisePodGetFractLvl(const PerlinNoisePod* const that);

// Get the fractal coefficient of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetFractCoeff(const PerlinNoisePod* const that);

// Get the border coefficient of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetBorder(const PerlinNoisePod* const that);

// Get the smoothness of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetSmooth(const PerlinNoisePod* const that);

// Get the squareness of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetSquare(const PerlinNoisePod* const that);

// Set the seed of the PerlinNoisePod 'that' to a copy of 'seed'
// If 'seed' is the vector null the seed is left unchanged
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetSeed(PerlinNoisePod* const that,
  const VecFloat3D* const seed);

// Set the bound of the PerlinNoisePod 'that' to 'bound'
// The Shapoid 'bound' must have same dimensions as the input dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetBound(PerlinNoisePod* const that,
  Shapoid* const bound);
```

```
// Set the scale of inputs of the PerlinNoisePod 'that' to a
// copy of 'scale'
// 'scale' must have same dimension as the input dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetScaleIn(PerlinNoisePod* const that,
  const VecFloat* const scale);

// Set the scale of outputs of the PerlinNoisePod 'that' to a copy
// of 'scale'
// 'scale' must have same dimension as the output dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetScaleOut(PerlinNoisePod* const that,
  const VecFloat* const scale);

// Set the shift of outputs of the PerlinNoisePod 'that' to a copy
// of 'shift'
// 'shift' must have same dimension as the output dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetShiftOut(PerlinNoisePod* const that,
  const VecFloat* const shift);

// Set the fractal level of the PerlinNoisePod 'that' to 'lvl'
// 'lvl' must be greater than or equal to 0
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetFractLvl(PerlinNoisePod* const that, const int lvl);

// Set the fractal coefficient of the PerlinNoisePod 'that' to 'coeff'
// 'coeff' must be greater than 0.0, if it's negative its absolute
// value is used
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetFractCoeff(PerlinNoisePod* const that,
  const float coeff);

// Set the border coefficient of the PerlinNoisePod 'that' to 'border'
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetBorder(PerlinNoisePod* const that,
  const float border);

// Set the smoothness of the PerlinNoisePod 'that' to 'smooth'
// 'smooth' must be greater than 0.0
// Below 1.0 gives a bumpy aspect, above 1.0 gives a thorny aspect,
// 1.0 is the smoothest
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetSmooth(PerlinNoisePod* const that,
```

```
  const float smooth);

// Set the squareness of the PerlinNoisePod 'that' to 'square'
// 'square' must be in [0.0, 1.0]
// 0.0 is the standard Perlin noise, 1.0 is the Perlin noise without
// the smoother function on input parameter
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetSquare(PerlinNoisePod* const that,
  const float square);

// Get the insideness in boundary of the PerlinNoisePod 'that' at
// position 'pos'
// If there is no bounday it's always 1.0
// If there is boundary, it's 0.0 outside of boundary and else it's
// the position depth in the Shapoid corrected with the border as follow:
// if depth > border the depth is set to 1.0, else the depth is
// SmootherStep from (0.0, border) to (0.0, 1.0)
// 'pos' 's dimension must be equal to the Shapoid's dimension
float PerlinNoisePodGetInsideness(PerlinNoisePod* const that,
  const VecFloat* const pos);

// -------------- FracNoise

typedef struct FracNoise {
  // Dimensions (input, output)
  const VecShort2D _dim;
  // Set of PerlinNoisePod
  GSet _noises;
} FracNoise;

// ================ Functions declaration ====================

// Return a new FracNoise of dimensions 'dim' and 'seed'
// The number of input must be between 1 and 3
// The number of output can be any value greater or equal to 1
// If seed is null it is replaced by a default seed
// Return null if the number of dimensions are invalid
FracNoise* FracNoiseCreate(const VecShort2D* const dim,
  const VecFloat3D* const seed);

// Free the memory used by the FracNoise 'that'
void FracNoiseFree(FracNoise** that);

// Get the dimensions of the noise 'that'
#if BUILDMODE != 0
inline
#endif
const VecShort2D* FracNoiseDim(const FracNoise* const that);

// Get the pods of the noise 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* FracNoisePods(const FracNoise* const that);

// Get the 'iNoise'-th noise of the FracNoise 'that'
// Return null if 'iNoise' is invalid
#if BUILDMODE != 0
inline
#endif
```

```
PerlinNoisePod* FracNoiseGetNoise(const FracNoise* const that,
  const int iNoise);

// Add a new noise with seed 'seed' to the FracNoise 'that'
// If seed is null it is replaced by a default seed
// Return the new noise
#if BUILDMODE != 0
inline
#endif
PerlinNoisePod* FracNoiseAddNoise(FracNoise* const that,
  const VecFloat3D* const seed);

// Remove the PerlinNoisePod 'pod' from the FracNoise 'that'
#if BUILDMODE != 0
inline
#endif
void FracNoiseRemoveNoise(FracNoise* const that,
  const PerlinNoisePod* const pod);

// Get the noise value of the FracNoise 'that' at 'pos'
VecFloat* _FracNoiseGet(const FracNoise* const that,
  const VecFloat* const pos);

// Export the FracNoise 'that' to a POV-Ray .df3 file located at
// 'filename'. The input range goes from 0.0 to 'range'. The number of
// samples per axis is given by 'nbSample'. The resolution in bit is
// given by 'res' (must be 8, 16 or 32).
// If 'rescale' equals true, the values are scaled to [0,1] else they
// are clipped
// The FracNoise's dimensions must be 3D->1D
// The output values are automatically scaled
// If the arguments are invalid, the df3 file is not generated.
// Return true if the df3 has been succesfully created, false else.
bool FracNoiseExportDF3(const FracNoise* const that,
  const VecFloat3D* const range, const VecShort3D* const nbSample,
  const int res, const bool rescale, const char* const fileName);

// ================ Polymorphism ====================
#define PerlinNoiseGet(Noise, U) _Generic(U, \
  VecFloat*: _PerlinNoiseGet, \
  const VecFloat*: _PerlinNoiseGet, \
  float: _PerlinNoiseGet1D, \
  const float: _PerlinNoiseGet1D, \
  VecFloat2D*: _PerlinNoiseGet2D, \
  const VecFloat2D*: _PerlinNoiseGet2D, \
  VecFloat3D*: _PerlinNoiseGet3D, \
  const VecFloat3D*: _PerlinNoiseGet3D, \
  default:PBErrInvalidPolymorphism) (Noise, U, 0.0)

#define PerlinNoisePodSetBound(Pod, Bound) _Generic(Bound, \
  Shapoid*: _PerlinNoisePodSetBound, \
  const Shapoid*: _PerlinNoisePodSetBound, \
  Facoid*: _PerlinNoisePodSetBound, \
  const Facoid*: _PerlinNoisePodSetBound, \
  Pyramidoid*: _PerlinNoisePodSetBound, \
  const Pyramidoid*: _PerlinNoisePodSetBound, \
  Spheroid*: _PerlinNoisePodSetBound, \
  const Spheroid*: _PerlinNoisePodSetBound, \
  default:PBErrInvalidPolymorphism) (Pod, (Shapoid*)Bound)

#define PerlinNoisePodSetScaleIn(Pod, Scale) _Generic(Scale, \
  VecFloat*: _PerlinNoisePodSetScaleIn, \
```

```
  const VecFloat*: _PerlinNoisePodSetScaleIn, \
  VecFloat2D*: _PerlinNoisePodSetScaleIn, \
  const VecFloat2D*: _PerlinNoisePodSetScaleIn, \
  VecFloat3D*: _PerlinNoisePodSetScaleIn, \
  const VecFloat3D*: _PerlinNoisePodSetScaleIn, \
  default:PBErrInvalidPolymorphism) (Pod, (const VecFloat*)(Scale))

#define PerlinNoisePodSetScaleOut(Pod, Scale) _Generic(Scale, \
  VecFloat*: _PerlinNoisePodSetScaleOut, \
  const VecFloat*: _PerlinNoisePodSetScaleOut, \
  VecFloat2D*: _PerlinNoisePodSetScaleOut, \
  const VecFloat2D*: _PerlinNoisePodSetScaleOut, \
  VecFloat3D*: _PerlinNoisePodSetScaleOut, \
  const VecFloat3D*: _PerlinNoisePodSetScaleOut, \
  default:PBErrInvalidPolymorphism) (Pod, (const VecFloat*)(Scale))

#define PerlinNoisePodSetShiftOut(Pod, Shift) _Generic(Shift, \
  VecFloat*: _PerlinNoisePodSetShiftOut, \
  const VecFloat*: _PerlinNoisePodSetShiftOut, \
  VecFloat2D*: _PerlinNoisePodSetShiftOut, \
  const VecFloat2D*: _PerlinNoisePodSetShiftOut, \
  VecFloat3D*: _PerlinNoisePodSetShiftOut, \
  const VecFloat3D*: _PerlinNoisePodSetShiftOut, \
  default:PBErrInvalidPolymorphism) (Pod, (const VecFloat*)(Shift))

#define FracNoiseGet(Noise, U) _Generic(U, \
  VecFloat*: _FracNoiseGet, \
  const VecFloat*: _FracNoiseGet, \
  VecFloat2D*: _FracNoiseGet, \
  const VecFloat2D*: _FracNoiseGet, \
  VecFloat3D*: _FracNoiseGet, \
  const VecFloat3D*: _FracNoiseGet, \
  default:PBErrInvalidPolymorphism) (Noise, (const VecFloat*)(U))

// ================ Inliner ====================

#if BUILDMODE != 0
#include "fracnoise-inline.c"
#endif

#endif
```

3 Code

3.1 fracnoise.c

```
// ============ FRACNOISE.C ================

// ================= Include =================

#include "fracnoise.h"
#if BUILDMODE == 0
  #include "fracnoise-inline.c"
#endif

// -------------- PerlinNoise

// Declare the permutations for the default Perlin noise
```

```
const int PerlinNoisePermutation[256] = {
  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
  };

// ================ Functions definition ====================

inline float fade(float t, float s);
inline float lerp(float t, float a, float b);
inline float grad(int hash, float x, float y, float z);

// ================ Functions implementation ====================

// Return a new PerlinNoise
PerlinNoise* PerlinNoiseCreate(void) {
  // Allocate memory
  PerlinNoise *noise = PBErrMalloc(FracNoiseErr,
    sizeof(PerlinNoise));
  // Initialize the permutations
  PerlinNoiseSetPermut(noise, PerlinNoisePermutation);
  // Return the new PerlinNoise
  return noise;
}

// Return a new static PerlinNoise
PerlinNoise PerlinNoiseCreateStatic(void) {
  // Declare the noise
  PerlinNoise noise;
  // Initialize the permutations
  PerlinNoiseSetPermut(&noise, PerlinNoisePermutation);
  // Return the noise
  return noise;
}

// Set the permutations of the PerlinNoise 'that' to 'permut'
// 'permut' is an array of 256 int
void PerlinNoiseSetPermut(PerlinNoise* const that,
  const int* const permut) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (permut == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'permut' is null");
    PBErrCatch(FracNoiseErr);
  }
```

```
#endif
  // Set the permutations
  for (int i = 256; i--;)
    that->_p[256 + i] = that->_p[i] = permut[i];
}

// Free the memory used by the PerlinNoise 'that'
void PerlinNoiseFree(PerlinNoise** that) {
  // Check arguments
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  free(*that);
  *that = NULL;
}

// Return the value of the PerlinNoise 'that' at position 'u'
// 'u' can be of dimension 1 to 3, 'u' values in R, the Perlin noise
// has interpolation on unit, so u in [0.0, 1.0] will be interpolated
// between 2 values, u in [0.0, 2.0] will be interpolated between
// 3 values, and the noise always equal 0.5 at unit values, so
// noise(0.0) == noise(1.0) == noise(2.0) == ... == 0.5
// Return a value in [0.0, 1.0]
float _PerlinNoiseGet(const PerlinNoise* const that,
  const VecFloat* const p, const float squareness) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (p == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'p' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGetDim(p) > 3) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'p' 's dimension is invalid (%d<4)",
      VecGetDim(p));
    PBErrCatch(FracNoiseErr);
  }
  if (squareness < 0.0 || squareness > 1.0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'square' is invalid (0<=%f<1)",
      squareness);
    PBErrCatch(FracNoiseErr);
  }
#endif
  if (VecGetDim(p) == 1)
    return _PerlinNoiseGet1D(that, VecGet(p, 0), squareness);
  else if (VecGetDim(p) == 2)
    return _PerlinNoiseGet2D(that, (VecFloat2D*)p, squareness);
  else if (VecGetDim(p) == 3)
    return _PerlinNoiseGet3D(that, (VecFloat3D*)p, squareness);
  else if (VecGetDim(p) > 3) {
    VecFloat* p3 = VecGetNewDim(p, 3);
    float ret = _PerlinNoiseGet3D(that, (VecFloat3D*)p3, squareness);
    VecFree(&p3);
    return ret;
  } else
    return 0.0;
```

```
}

float _PerlinNoiseGet1D(const PerlinNoise* const that, const float p,
  const float squareness) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (squareness < 0.0 || squareness > 1.0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'square' is invalid (0<=%f<1)",
      squareness);
    PBErrCatch(FracNoiseErr);
  }
#endif
  int pInt;
  float q = 0.0;
  float u = 0.0;
  // Get the Unit cube that contains the point
  pInt = (int)floor(p) & 255;
  // Get the relative coordinates of the point in the cube
  q = p - floor(p);
  // Fade input for smooth transition
  u = fade(q, squareness);
  // Hash coordinates of the segment extremities
  int A = that->_p[pInt];
  int AA = that->_p[A];
  int B = that->_p[pInt + 1];
  int BA = that->_p[B];
  // Blend the value of the 8 cube corners according to relative
  // position in the cube
  float res = lerp(u, grad(that->_p[AA], q, 0.0, 0.0),
    grad(that->_p[BA], q - 1.0, 0.0, 0.0));
  // Return the result mapped to [0.0, 1.0]
  res = res * 0.5 + 0.5;
  return res;
}

float _PerlinNoiseGet2D(const PerlinNoise* const that,
  const VecFloat2D* const p, float squareness) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (p == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'p' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGetDim(p) != 2) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'p' 's dimension is invalid (%d==2)",
      VecGetDim(p));
    PBErrCatch(FracNoiseErr);
  }
  if (squareness < 0.0 || squareness > 1.0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'square' is invalid (0<=%f<1)",
```

```
      squareness);
    PBErrCatch(FracNoiseErr);
  }
#endif
  int pInt[2];
  VecFloat2D q = VecFloatCreateStatic2D();
  VecFloat2D u = VecFloatCreateStatic2D();
  for (int i = 2; i--;) {
    // Get the Unit cube that contains the point
    pInt[i] = (int)floor(VecGet(p, i)) & 255;
    // Get the relative coordinates of the point in the cube
    VecSet(&q, i, VecGet(p, i) - floor(VecGet(p, i)));
    // Fade input for smooth transition
    VecSet(&u, i, fade(VecGet(&q, i), squareness));
  }
  // Hash coordinates of the 8 cube corners
  int A = that->_p[pInt[0]] + pInt[1];
  int AA = that->_p[A];
  int AB = that->_p[A + 1];
  int B = that->_p[pInt[0] + 1] + pInt[1];
  int BA = that->_p[B];
  int BB = that->_p[B + 1];
  // Blend the value of the 8 cube corners according to relative
  // position in the cube
  float res =
    lerp(0.0,
      lerp(VecGet(&u, 1),
        lerp(VecGet(&u, 0),
          grad(that->_p[AA],
            VecGet(&q, 0), VecGet(&q, 1), 0.0),
          grad(that->_p[BA],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1), 0.0)),
        lerp(VecGet(&u, 0),
          grad(that->_p[AB],
            VecGet(&q, 0), VecGet(&q, 1) - 1.0, 0.0),
          grad(that->_p[BB],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1) - 1.0, 0.0))),
      lerp(VecGet(&u, 1),
        lerp(VecGet(&u, 0),
          grad(that->_p[AA + 1],
            VecGet(&q, 0), VecGet(&q, 1), -1.0),
          grad(that->_p[BA + 1],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1), -1.0)),
        lerp(VecGet(&u, 0),
          grad(that->_p[AB + 1],
            VecGet(&q, 0), VecGet(&q, 1) - 1.0, -1.0),
          grad(that->_p[BB + 1],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1) - 1.0, -1.0))));
  // Return the result mapped to [0.0, 1.0]
  res = res * 0.5 + 0.5;
  return res;
}

float _PerlinNoiseGet3D(const PerlinNoise* const that,
  const VecFloat3D* const p, const float squareness) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (p == NULL) {
```

```
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'p' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGetDim(p) != 3) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'p' 's dimension is invalid (%d==3)",
      VecGetDim(p));
    PBErrCatch(FracNoiseErr);
  }
  if (squareness < 0.0 || squareness > 1.0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'squareness' is invalid (0<=%f<1)",
      squareness);
    PBErrCatch(FracNoiseErr);
  }
#endif
  int pInt[3];
  VecFloat3D q = VecFloatCreateStatic3D();
  VecFloat3D u = VecFloatCreateStatic3D();
  for (int i = 3; i--;) {
    // Get the Unit cube that contains the point
    pInt[i] = (int)floor(VecGet(p, i)) & 255;
    // Get the relative coordinates of the point in the cube
    VecSet(&q, i, VecGet(p, i) - floor(VecGet(p, i)));
    // Fade input for smooth transition
    VecSet(&u, i, fade(VecGet(&q, i), squareness));
  }
  // Hash coordinates of the 8 cube corners
  int A = that->_p[pInt[0]] + pInt[1];
  int AA = that->_p[A] + pInt[2];
  int AB = that->_p[A + 1] + pInt[2];
  int B = that->_p[pInt[0] + 1] + pInt[1];
  int BA = that->_p[B] + pInt[2];
  int BB = that->_p[B + 1] + pInt[2];
  // Blend the value of the 8 cube corners according to relative
  // position in the cube
  float res =
    lerp(VecGet(&u, 2),
      lerp(VecGet(&u, 1),
        lerp(VecGet(&u, 0),
          grad(that->_p[AA],
            VecGet(&q, 0), VecGet(&q, 1), VecGet(&q, 2)),
          grad(that->_p[BA],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1), VecGet(&q, 2))),
        lerp(VecGet(&u, 0),
          grad(that->_p[AB],
            VecGet(&q, 0), VecGet(&q, 1) - 1.0, VecGet(&q, 2)),
          grad(that->_p[BB],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1) - 1.0, VecGet(&q, 2)))),
      lerp(VecGet(&u, 1),
        lerp(VecGet(&u, 0),
          grad(that->_p[AA + 1],
            VecGet(&q, 0), VecGet(&q, 1), VecGet(&q, 2) - 1.0),
          grad(that->_p[BA + 1],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1), VecGet(&q, 2) - 1.0)),
        lerp(VecGet(&u, 0),
          grad(that->_p[AB + 1],
            VecGet(&q, 0), VecGet(&q, 1) - 1.0, VecGet(&q, 2) - 1.0),
          grad(that->_p[BB + 1],
            VecGet(&q, 0) - 1.0, VecGet(&q, 1) - 1.0,
            VecGet(&q, 2) - 1.0))));
```

```
  // Return the result mapped to [0.0, 1.0]
  res = res * 0.5 + 0.5;
  return res;
}

inline float fade(float t, float s) {
  float ret =
    (1.0 - s) * t * t * t * (t * (t * 6.0 - 15.0) + 10.0) + s * t;
  return ret;
}

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

float grad(int hash, float x, float y, float z) {
  int h = hash & 15;
  float u = h < 8 ? x : y;
  float v = h < 4 ? y : h == 12 || h == 14 ? x : z;
  return ((h & 1) == 0 ? u : -u) + ((h & 2) == 0 ? v : -v);
}

// -------------- PerlinNoisePod

// ================ Functions implementation ====================

// Return a new PerlinNoisePod with output dimension 'dimOut' and seed
// 'seed'
// If the seed is the vector null replace it by (1,1,1)
// Default values:
// bound = NULL, border = 0.1, scaleIn/Out = 1.0
// fractalLvl = 0, fractalCoeff = 0.1, smooth = 1.0
PerlinNoisePod* PerlinNoisePodCreate(const VecShort2D* const dim,
  const VecFloat3D* const seed) {
#if BUILDMODE == 0
  if (VecGet(dim, 0) < 1 || VecGet(dim, 0) > 3 ||
    VecGet(dim, 1) < 1) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'dim' is invalid (0<%d<4,0<%d)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Declare the new PerlinNoisePod
  PerlinNoisePod* pod = PBErrMalloc(FracNoiseErr,
    sizeof(PerlinNoisePod));
  // Set properties
  pod->_noise = PerlinNoiseCreateStatic();
  // If the seed is not given or is the vector null replace it by (1,1,1)
  if (seed == NULL || VecNorm(seed) < PBMATH_EPSILON) {
    pod->_seed = VecFloatCreateStatic3D();
    for (int i = 3; i--;)
      VecSet(&(pod->_seed), i, 1.0);
  // Else, copy the seed in argument
  } else {
    pod->_seed = *seed;
  }
  // Ensure the seed is normalized
  VecNormalise(&(pod->_seed));
  pod->_bound = NULL;
  pod->_border = 0.1;
```

```
  pod->_smooth = 1.0;
  pod->_square = 0.0;
  pod->_scaleIn = VecFloatCreate(VecGet(dim, 0));
  for (int i = VecGet(dim, 0); i--;)
    VecSet(pod->_scaleIn, i, 1.0);
  pod->_scaleOut = VecFloatCreate(VecGet(dim, 1));
  for (int i = VecGet(dim, 1); i--;)
    VecSet(pod->_scaleOut, i, 1.0);
  pod->_shiftOut = VecFloatCreate(VecGet(dim, 1));
  pod->_fractalLvl = 0;
  pod->_fractalCoeff = 0.1;
  // Return the new pod
  return pod;
}

// Free memory used by the PerlinNoisePod 'that'
void PerlinNoisePodFree(PerlinNoisePod** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  VecFree(&((*that)->_scaleIn));
  VecFree(&((*that)->_scaleOut));
  VecFree(&((*that)->_shiftOut));
  free(*that);
  *that = NULL;
}

// Get the insideness in boundary of the PerlinNoisePod 'that' at
// position 'pos'
// If there is no bounday it's always 1.0
// If there is boundary, it's 0.0 outside of boundary and else it's
// the position depth in the Shapoid corrected with the border as follow:
// if depth > border the depth is set to 1.0, else the depth is
// fade from (0.0, border) to (0.0, 1.0)
// 'pos' 's dimension must be equal to the Shapoid's dimension
float PerlinNoisePodGetInsideness(PerlinNoisePod* const that,
  const VecFloat* const pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (pos == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'pos' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (that->_bound != NULL &&
    VecGetDim(pos) != ShapoidGetDim(that->_bound)) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'pos' 's dimension is invalid (%d=%d)",
      VecGetDim(pos), ShapoidGetDim(that->_bound));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Declare a variable to memorize the insideness in boundary
  float inside = 1.0;
  // If there is a boundary
```

```
  if (PerlinNoisePodBound(that) != NULL) {
    // Get the insideness of the input
    inside = ShapoidGetPosDepth(PerlinNoisePodBound(that), pos);
    // If there is a border
    float border = PerlinNoisePodGetBorder(that);
    if (border > PBMATH_EPSILON) {
      // Correct insideness with border: inside the border apply a
      // smootherstep transition and outside set to 1.0
      if (inside < border)
        inside = fade(inside / border, 0.0);
      else
        inside = 1.0;
    }
  }
  // Return the insideness
  return inside;
}

// -------------- FracNoise

// ================ Functions implementation ====================

// Return a new FracNoise of dimensions 'dim' and seed 'seed'
// The number of input must be between 1 and 3
// The number of output can be any value greater or equal to 1
// If seed is null it is replaced by a default seed
// Return null if the number of dimensions are invalid
FracNoise* FracNoiseCreate(const VecShort2D* const dim,
  const VecFloat3D* const seed) {
#if BUILDMODE == 0
  if (dim == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'dim' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGet(dim, 0) < 1 || VecGet(dim, 0) > 3 ||
    VecGet(dim, 1) < 1) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'dim' is invalid (0<%d<4,0<%d)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Check the dimensions
  if (VecGet(dim, 0) < 1 || VecGet(dim, 0) > 3 ||
    VecGet(dim, 1) < 1)
    return NULL;
  // Declare the new FracNoise
  FracNoise* noise = PBErrMalloc(FracNoiseErr, sizeof(FracNoise));
  // Set properties
  *(VecShort2D*)&(noise->_dim) = VecShortCreateStatic2D();
  *(VecShort2D*)&(noise->_dim) = *dim;
  noise->_noises = GSetCreateStatic();
  // Add a PerlinNoise
  PerlinNoisePod* pod = PerlinNoisePodCreate(dim, seed);
  GSetAppend(&(noise->_noises), pod);
  // Return the new FracNoise
  return noise;
}

// Free the memory used by the FracNoise 'that'
void FracNoiseFree(FracNoise** that) {
```

```
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  while (GSetNbElem(&((*that)->_noises)) > 0) {
    PerlinNoisePod* pod = GSetPop(&((*that)->_noises));
    PerlinNoisePodFree(&pod);
  }
  free(*that);
  *that = NULL;
}

// Get the noise value of the FracNoise 'that' at 'pos'
VecFloat* _FracNoiseGet(const FracNoise* const that,
  const VecFloat* const pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (pos == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'pos' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGetDim(pos) != VecGet(&(that->_dim), 0)) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'pos' 's dimension is invalid (%d==%d)",
      VecGetDim(pos), VecGet(&(that->_dim), 0));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Create the result
  VecFloat* res = VecFloatCreate(VecGet(FracNoiseDim(that), 1));
  // Loop on noises
  GSetIterForward iter =
    GSetIterForwardCreateStatic(FracNoisePods(that));
  do {
    // Get the current pod
    PerlinNoisePod* pod = GSetIterGet(&iter);
    // Get the insideness
    float inside = PerlinNoisePodGetInsideness(pod, pos);
    // If we are inside the boundary
    if (inside > PBMATH_EPSILON) {
      // Internally FracNoise always considers the input position as 3D,
      // declare a 3D version of it
      VecFloat3D* p = (VecFloat3D*)VecGetNewDim(pos, 3);
      // Scale the input
      for (int i = MIN(3, VecGetDim(pos)); i--;)
        VecSet(p, i,
           VecGet(p, i) * VecGet(PerlinNoisePodScaleIn(pod), i));
      // Declare a variable to memorize the rotated version of the input
      VecFloat3D pRot = VecFloatCreateStatic3D();
      // Declare a variable to memorize the result for this pod
      VecFloat* resPod = VecFloatCreate(VecGet(FracNoiseDim(that), 1));
      // Declare a variable to memorize the current fractal coefficient
      float fractCoeff = 1.0;
      // Declare a variable to memorize the fractal coeff
      float fc = PerlinNoisePodGetFractCoeff(pod);
```

```
      // Get the angle for rotation of input vector over dimensions
      float theta =
        PBMATH_TWOPI / (float)(VecGet(FracNoiseDim(that), 1) + 1);
      // Loop on fractal levels
      for (int iFract = PerlinNoisePodGetFractLvl(pod) + 1; iFract--;) {
        // If we are not at the first fractal level
        if (iFract != PerlinNoisePodGetFractLvl(pod))
          // Rescale to ensure the values stay in [0.0, 1.0] as addition
          // over fractal levels may bring the value outside this range
          VecScale(resPod, 1.0 / (1.0 + fractCoeff));
        // Loop on output dimension
        VecCopy(&pRot, p);
        for (int iDim = VecGet(FracNoiseDim(that), 1); iDim--;) {
          // Apply the seed
          VecRotAxis(&pRot, PerlinNoisePodSeed(pod), theta);
          // Get the Perlin noise value
          float val = _PerlinNoiseGet3D(PerlinNoisePodNoise(pod), &pRot,
            PerlinNoisePodGetSquare(pod));
          // Add the value for the current dimension at the current
          // fractal level
          VecSet(resPod, iDim, VecGet(resPod, iDim) + val * fractCoeff);
        }
        // Update the fractal coefficient
        fractCoeff *= fc;
        // Update the input position at this level
        VecScale(p, 1.0 / fc);
      }
      // Apply the smoothness
      for (int i = VecGetDim(resPod); i--;)
        VecSet(resPod, i,
          pow(VecGet(resPod, i), PerlinNoisePodGetSmooth(pod)));
      // Scale the output
      for (int i = VecGetDim(resPod); i--;)
        VecSet(resPod, i,
          VecGet(resPod, i) * VecGet(PerlinNoisePodScaleOut(pod), i));
      // Shift the output
      VecOp(resPod, 1.0, PerlinNoisePodShiftOut(pod), 1.0);
      // Apply the insideness
      for (int i = VecGetDim(resPod); i--;)
        VecSet(resPod, i, VecGet(resPod, i) * inside);
      // Add the result for this pod
      VecOp(res, 1.0, resPod, 1.0);
      // Free memory
      VecFree(&resPod);
      VecFree((VecFloat**)&p);
    }
  } while (GSetIterStep(&iter));
  // Return the result
  return res;
}

// Export the FracNoise 'that' to a POV-Ray .df3 file located at
// 'filename'. The input range goes from 0.0 to 'range'. The number of
// samples per axis is given by 'nbSample'. The resolution in bit is
// given by 'res' (must be 8, 16 or 32).
// If 'rescale' equals true, the values are scaled to [0,1] else they
// are clipped
// The FracNoise's dimensions must be 3D->1D
// The output values are automatically scaled
// If the arguments are invalid, the df3 file is not generated.
// Return true if the df3 has been succesfully created, false else.
bool FracNoiseExportDF3(const FracNoise* const that,
```

```
  const VecFloat3D* const range, const VecShort3D* const nbSample,
  const int res, const bool rescale, const char* const fileName) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (range == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'range' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (nbSample == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'nbSample' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (fileName == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'fileName' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGet(FracNoiseDim(that), 0) != 3) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'that' 's input dimension is invalid (%d==3)",
      VecGet(FracNoiseDim(that), 0));
    PBErrCatch(FracNoiseErr);
  }
  if (VecGet(FracNoiseDim(that), 1) != 1) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'that' 's output dimension is invalid (%d==1)",
      VecGet(FracNoiseDim(that), 1));
    PBErrCatch(FracNoiseErr);
  }
  if (VecGet(nbSample, 0) <= 0 ||
    VecGet(nbSample, 1) <= 0 ||
    VecGet(nbSample, 2) <= 0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'nbSample' is invalid (<%d,%d,%d> > <0,0,0>)",
      VecGet(nbSample, 0), VecGet(nbSample, 1), VecGet(nbSample, 2));
    PBErrCatch(FracNoiseErr);
  }
  if (res != 8 && res != 16 && res != 32) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'res' is invalid (%d=={8,16,32})", res);
    PBErrCatch(FracNoiseErr);
  }
  if (GSetNbElem(&(that->_noises)) == 0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'that' is empty");
    PBErrCatch(FracNoiseErr);
  }
  if (sizeof(unsigned char) != 1 ||
    sizeof(unsigned short) != 2 ||
    sizeof(unsigned int) != 4) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
```

```
      "types' size mismatched (%d==1,%d==2,%d==4)",
      sizeof(unsigned char),sizeof(unsigned short),
      sizeof(unsigned int));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Check arguments
  if (VecGet(FracNoiseDim(that), 0) != 3 ||
    VecGet(FracNoiseDim(that), 1) != 1 ||
    VecGet(nbSample, 0) <= 0 ||
    VecGet(nbSample, 1) <= 0 ||
    VecGet(nbSample, 2) <= 0 ||
    (res != 8 && res != 16 && res != 32) ||
    GSetNbElem(&(that->_noises)) == 0)
    return false;
  // Get the min and max values of the noise for automatic rescaling
  float min = 0.0;
  float max = 1.0;
  if (rescale) {
    PerlinNoisePod* pod = FracNoiseGetNoise(that, 0);
    if (VecGet(PerlinNoisePodScaleOut(pod), 0) > 0.0) {
      min = VecGet(PerlinNoisePodShiftOut(pod), 0);
      max = min + VecGet(PerlinNoisePodScaleOut(pod), 0);
    } else {
      max = VecGet(PerlinNoisePodShiftOut(pod), 0);
      min = max + VecGet(PerlinNoisePodScaleOut(pod), 0);
    }
    for (int iNoise = 1; iNoise < GSetNbElem(&(that->_noises));
      ++iNoise) {
      pod = FracNoiseGetNoise(that, iNoise);
      float minP = 0.0;
      float maxP = 1.0;
      if (VecGet(PerlinNoisePodScaleOut(pod), 0) > 0.0) {
        minP = VecGet(PerlinNoisePodShiftOut(pod), 0);
        maxP = minP + VecGet(PerlinNoisePodScaleOut(pod), 0);
      } else {
        maxP = VecGet(PerlinNoisePodShiftOut(pod), 0);
        minP = maxP + VecGet(PerlinNoisePodScaleOut(pod), 0);
      }
      if (minP < 0.0)
        min += minP;
      if (maxP > 0.0)
        max += maxP;
    }
  }
  // Open the file
  FILE* file = fopen(fileName, "wb");
  // If we couldn't open the file
  if (file == NULL)
    // Stop here
    return false;
  // Write the header
  for (int i = 0; i < 3; ++i) {
    unsigned short v = VecGet(nbSample, i);
    // Big-endian !
    char* ptr = (char*)&v;
    for (int ip = sizeof(unsigned short); ip--;)
      fwrite(ptr + ip, 1, 1, file);
  }
  // Write the body
  // Loop on position
  VecShort3D pos = VecShortCreateStatic3D();
```

```
  do {
    // Get the equivalent input
    VecFloat3D u = VecFloatCreateStatic3D();
    for (int i = 3; i--;)
      VecSet(&u, i,
        (float)VecGet(&pos, i) / (float)VecGet(nbSample, i) *
        VecGet(range, i));
    // Get the noise value
    VecFloat* out = FracNoiseGet(that, &u);
    float val = VecGet(out, 0);
    VecFree(&out);
    // Scale or clip the output value
    if (rescale)
      val = (val - min) / (max - min);
    if (val < 0.0)
      val = 0.0;
    else if (val > 1.0)
      val = 1.0;
    // Write the value according to resolution
    if (res == 8) {
      unsigned char vChar = (unsigned char)round(val * 255.0);
      fwrite(&vChar, sizeof(unsigned char), 1, file);
    } else if (res == 16) {
      unsigned short vShort = (unsigned short)round(val * 65535.0);
      // Big-endian !
      char* ptr = (char*)&vShort;
      for (int ip = sizeof(unsigned short); ip--;)
        fwrite(ptr + ip, 1, 1, file);
    } else if (res == 32) {
      unsigned int vInt = (unsigned int)round(val * 4294967296.0);
      // Big-endian !
      char* ptr = (char*)&vInt;
      for (int ip = sizeof(unsigned int); ip--;)
        fwrite(ptr + ip, 1, 1, file);
    }
  // The data in df3 are stored as z,y,x -> PStep
  } while (VecPStep(&pos, nbSample));
  // Close the file
  fclose(file);
  // Return the success code
  return true;
}
```

3.2 fracnoise-inline.c

```
// ============ FRACNOISE-INLINE.C ================

// -------------- PerlinNoisePod

// ================ Functions implementation ====================

// Get the Perlin noise of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
const PerlinNoise* PerlinNoisePodNoise(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return &(that->_noise);
}

// Get the seed of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat3D* PerlinNoisePodSeed(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return (VecFloat3D*)&(that->_seed);
}

// Get the bound of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
Shapoid* PerlinNoisePodBound(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return that->_bound;
}

// Get the scale of inputs of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* PerlinNoisePodScaleIn(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return that->_scaleIn;
}

// Get the scale of outputs of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* PerlinNoisePodScaleOut(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
```

```
  }
#endif
  return that->_scaleOut;
}

// Get the shift of outputs of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* PerlinNoisePodShiftOut(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return that->_shiftOut;
}

// Get the fractal level of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
int PerlinNoisePodGetFractLvl(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return that->_fractalLvl;
}

// Get the fractal coefficient of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetFractCoeff(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return that->_fractalCoeff;
}

// Get the border coefficient of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetBorder(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
```

```
  return that->_border;
}

// Get the smoothness of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetSmooth(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return that->_smooth;
}

// Get the squareness of the PerlinNoisePod 'that'
#if BUILDMODE != 0
inline
#endif
float PerlinNoisePodGetSquare(const PerlinNoisePod* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return that->_square;
}

// Set the seed of the PerlinNoisePod 'that' to a copy of 'seed'
// If 'seed' is the vector null the seed is left unchanged
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetSeed(PerlinNoisePod* const that,
  const VecFloat3D* const seed) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (seed == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'seed' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Check if the seed is the vector null
  if (VecNorm(seed) < PBMATH_EPSILON)
    return;
  // Set the new seed
  that->_seed = *seed;
  // Make sure the seed is normalised
  VecNormalise(&(that->_seed));
}

// Set the bound of the PerlinNoisePod 'that' to 'bound'
```

```
// The Shapoid 'bound' must have same dimensions as the input dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetBound(PerlinNoisePod* const that,
  Shapoid* const bound) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (bound == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'bound' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (ShapoidGetDim(bound) != VecGetDim(that->_scaleIn)) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'bound' 's dimension is invalid (%d=%d)",
      ShapoidGetDim(bound), VecGetDim(that->_scaleIn));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new bound
  that->_bound = bound;
}

// Set the scale of inputs of the PerlinNoisePod 'that' to a
// copy of 'scale'
// 'scale' must have same dimension as the input dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetScaleIn(PerlinNoisePod* const that,
  const VecFloat* const scale) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (scale == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'scale' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGetDim(scale) != VecGetDim(that->_scaleIn)) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'scale' 's dimension is invalid (%d=%d)",
      VecGetDim(scale), VecGetDim(that->_scaleIn));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new scale
  VecCopy(that->_scaleIn, scale);
}
```

```
// Set the scale of outputs of the PerlinNoisePod 'that' to a copy
// of 'scale'
// 'scale' must have same dimension as the output dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetScaleOut(PerlinNoisePod* const that, const VecFloat* const scale) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (scale == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'scale' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGetDim(scale) != VecGetDim(that->_scaleOut)) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'scale' 's dimension is invalid (%d=%d)",
      VecGetDim(scale), VecGetDim(that->_scaleOut));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new scale
  VecCopy(that->_scaleOut, scale);
}

// Set the shift of outputs of the PerlinNoisePod 'that' to a copy
// of 'shift'
// 'shift' must have same dimension as the output dimension
// of the PerlinNoisePod.
#if BUILDMODE != 0
inline
#endif
void _PerlinNoisePodSetShiftOut(PerlinNoisePod* const that,
  const VecFloat* const shift) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (shift == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'shift' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (VecGetDim(shift) != VecGetDim(that->_shiftOut)) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg,
      "'shift' 's dimension is invalid (%d=%d)",
      VecGetDim(shift), VecGetDim(that->_shiftOut));
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new shift
  VecCopy(that->_shiftOut, shift);
}
```

```
// Set the fractal level of the PerlinNoisePod 'that' to 'lvl'
// 'lvl' must be greater than or equal to 0
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetFractLvl(PerlinNoisePod* const that, const int lvl) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (lvl < 0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'lvl' is invalid (%d>=0)", lvl);
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new level
  that->_fractalLvl = lvl;
}

// Set the fractal coefficient of the PerlinNoisePod 'that' to 'coeff'
// 'coeff' must be greater than 0.0, if it's negative its absolute
// value is used
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetFractCoeff(PerlinNoisePod* const that,
  const float coeff) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (fabs(coeff) < PBMATH_EPSILON) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'coeff' is equal to 0.0");
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new coefficient
  that->_fractalCoeff = fabs(coeff);
}

// Set the fractal coefficient of the PerlinNoisePod 'that' to 'border'
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetBorder(PerlinNoisePod* const that,
  const float border) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new coefficient
  that->_border = border;
```

```
}

// Set the smoothness of the PerlinNoisePod 'that' to 'smooth'
// 'smooth' must be greater than 0.0
// Below 1.0 gives a bumpy aspect, above 1.0 gives a spiky aspect,
// 1.0 is the smoothest
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetSmooth(PerlinNoisePod* const that,
  const float smooth) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (smooth < PBMATH_EPSILON) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'smooth' is invalid (%f>0)", smooth);
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new coefficient
  that->_smooth = smooth;
}

// Set the squareness of the PerlinNoisePod 'that' to 'square'
// 'square' must be in [0.0, 1.0]
// 0.0 is the standard Perlin noise, 1.0 is the Perlin noise without
// the smoother function on input parameter
#if BUILDMODE != 0
inline
#endif
void PerlinNoisePodSetSquare(PerlinNoisePod* const that,
  const float square) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
  if (square < 0.0 || square > 1.0) {
    FracNoiseErr->_type = PBErrTypeInvalidArg;
    sprintf(FracNoiseErr->_msg, "'square' is invalid (0<=%f<1)", square);
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Set the new coefficient
  that->_square = square;
}

// -------------- FracNoise

// ================ Functions implementation ====================

// Get the dimensions of the noise 'that'
#if BUILDMODE != 0
inline
#endif
const VecShort2D* FracNoiseDim(const FracNoise* const that) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return &(that->_dim);
}

// Get the pods of the noise 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* FracNoisePods(const FracNoise* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  return &(that->_noises);
}


// Get the 'iNoise'-th noise of the FracNoise 'that'
// Return null if 'iNoise' is invalid
#if BUILDMODE != 0
inline
#endif
PerlinNoisePod* FracNoiseGetNoise(const FracNoise* const that,
  const int iNoise) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  if (iNoise < 0 || iNoise >= GSetNbElem(&(that->_noises)))
    return NULL;
  return (PerlinNoisePod*)GSetGet(&(that->_noises), iNoise);
}

// Add a new noise with seed 'seed' to the FracNoise 'that'
// If seed is null it is replaced by a default seed
// Return the new noise
#if BUILDMODE != 0
inline
#endif
PerlinNoisePod* FracNoiseAddNoise(FracNoise* const that,
  const VecFloat3D* const seed) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Create a new pod
  PerlinNoisePod* pod = PerlinNoisePodCreate(FracNoiseDim(that), seed);
  // Add the pod to the set of noises
```

```
  GSetAppend(&(that->_noises), pod);
  // Return the new pod
  return pod;
}

// Remove the PerlinNoisePod 'pod' from the FracNoise 'that'
#if BUILDMODE != 0
inline
#endif
void FracNoiseRemoveNoise(FracNoise* const that,
  const PerlinNoisePod* const pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    FracNoiseErr->_type = PBErrTypeNullPointer;
    sprintf(FracNoiseErr->_msg, "'that' is null");
    PBErrCatch(FracNoiseErr);
  }
#endif
  // Loop on the set of noises
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_noises));
  bool flag = true;
  do {
    // Get the current pod
    PerlinNoisePod* p = GSetIterGet(&iter);
    // If it's the searched pod
    if (p == pod) {
      // Remove from the set
      GSetIterRemoveElem(&iter);
      // Free memory
      PerlinNoisePodFree(&p);
      // End the loop on the set
      flag = false;
    }
  } while (flag && GSetIterStep(&iter));
}
```

4 Makefile

```
#directory
PBERRDIR=../PBErr
PBMATHDIR=../PBMath
GENBRUSHDIR=../GenBrush
GSETDIR=../GSet
SHAPOIDDIR=../Shapoid
BCURVEDIR=../BCurve
GTREEDIR=../GTree
PBJSONDIR=../PBJson

# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILDMODE=1

# 0: monolith version, the GBSurface is rendered toward a TGA image
# 1: GTK version, the GBSurface is rendered toward a TGA image or
#    a GtkWidget
```

```
BUILDWITHGRAPHICLIB=0

include $(PBERRDIR)/Makefile.inc

INCPATH=-I./ -I$(PBERRDIR)/ -I$(PBMATHDIR)/ -I$(GENBRUSHDIR)/ -I$(SHAPOIDDIR)/ -I$(BCURVEDIR)/ -I$(GSETDIR)/ -I$(PBJS
BUILDOPTIONS=$(BUILDPARAM) $(INCPATH) -ggdb

# compiler
COMPILER=gcc

#rules
all : main

main: main.o pberr.o bcurve.o shapoid.o pbmath.o gset.o genbrush.o fracnoise.o pbjson.o gtree.o Makefile
$(COMPILER) main.o pberr.o bcurve.o shapoid.o pbmath.o gset.o pbjson.o gtree.o genbrush.o fracnoise.o $(LINKOPTIONS)

main.o : main.c $(PBERRDIR)/pberr.h $(SHAPOIDDIR)/shapoid.h $(SHAPOIDDIR)/shapoid-inline.c $(PBMATHDIR)/pbmath.h $(PB
$(COMPILER) $(BUILDOPTIONS) -c main.c

pbjson.o : $(PBJSONDIR)/pbjson.c $(PBJSONDIR)/pbjson-inline.c $(PBJSONDIR)/pbjson.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBJSONDIR)/pbjson.c

gtree.o : $(GTREEDIR)/gtree.c $(GTREEDIR)/gtree.h $(GTREEDIR)/gtree-inline.c Makefile $(GSETDIR)/gset-inline.c $(GSET
$(COMPILER) $(BUILDOPTIONS) -c $(GTREEDIR)/gtree.c

fracnoise.o: fracnoise.h fracnoise.c fracnoise-inline.c $(PBMATHDIR)/pbmath.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c fracnoise.c

genbrush.o : $(GENBRUSHDIR)/genbrush.c $(GENBRUSHDIR)/genbrush.h $(GENBRUSHDIR)/genbrush-inline.c $(BCURVEDIR)/bcurve
$(COMPILER) $(BUILDOPTIONS) -c $(GENBRUSHDIR)/genbrush.c

bcurve.o : $(BCURVEDIR)/bcurve.c $(BCURVEDIR)/bcurve-inline.c $(BCURVEDIR)/bcurve.h $(SHAPOIDDIR)/shapoid.h $(SHAPOID
$(COMPILER) $(BUILDOPTIONS) -c $(BCURVEDIR)/bcurve.c

shapoid.o : $(SHAPOIDDIR)/shapoid.c $(SHAPOIDDIR)/shapoid.h $(SHAPOIDDIR)/shapoid-inline.c $(PBMATHDIR)/pbmath.h $(PB
$(COMPILER) $(BUILDOPTIONS) -c $(SHAPOIDDIR)/shapoid.c

pberr.o : $(PBERRDIR)/pberr.c $(PBERRDIR)/pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBERRDIR)/pberr.c

pbmath.o : $(PBMATHDIR)/pbmath.c $(PBMATHDIR)/pbmath-inline.c $(PBMATHDIR)/pbmath.h Makefile $(PBERRDIR)/pberr.h
$(COMPILER) $(BUILDOPTIONS) -c $(PBMATHDIR)/pbmath.c

gset.o : $(GSETDIR)/gset.c $(GSETDIR)/gset-inline.c $(GSETDIR)/gset.h Makefile $(PBERRDIR)/pberr.h
$(COMPILER) $(BUILDOPTIONS) -c $(GSETDIR)/gset.c

clean :
rm -rf *.o main

valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main

unitTest :
main > unitTest.txt; diff unitTest.txt unitTestRef.txt

HF:
povray +FT -W500 -H300 -D -P -Q9 +A +UA -IHF.pov

DF3:
povray +FT -W500 -H300 -D -P -Q9 +A -IDF3.pov
```

5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "fracnoise.h"
#include "genbrush.h"

#define RANDOMSEED 0

void UnitTestPerlinNoise() {
  // Create a PerlinNoise
  PerlinNoise* noise = PerlinNoiseCreate();
  PerlinNoise noiseStatic = PerlinNoiseCreateStatic();
  // Create a GB to render the image
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 512); VecSet(&dim, 1, 512);
  GenBrush* gb = GBCreateImage(&dim);
  // Draw the Image
  VecShort2D pos = VecShortCreateStatic2D();
  VecFloat3D p = VecFloatCreateStatic3D();
  VecSet(&p, 2, 0.4);
  GBPixel pix = GBColorBlack;
  do {
    for (int i = 2; i--;)
      VecSet(&p, i, (float)VecGet(&pos, i) / 64.0);
    float pn = PerlinNoiseGet(noise, &p);
    float pnStatic = PerlinNoiseGet(&noiseStatic, &p);
    if (ISEQUALF(pn, pnStatic) == false) {
      FracNoiseErr->_type = PBErrTypeUnitTestFailed;
      sprintf(FracNoiseErr->_msg, "PerlinNoiseStatic NOK");
      PBErrCatch(FracNoiseErr);
    }
    unsigned char n = (unsigned char)floor(255.0 * pn);
    pix._rgba[GBPixelRed] = n;
    pix._rgba[GBPixelGreen] = n;
    pix._rgba[GBPixelBlue] = n;
    GBSetFinalPixel(gb, &pos, &pix);
  } while (VecStep(&pos, &dim));
  // Compare to reference
  GenBrush* gbRef = GBCreateFromFile("./UnitTestPerlinNoise3DRef.tga");
  if (gbRef == NULL) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg,
      "Couln't load UnitTestPerlinNoise3DRef.tga");
    PBErrCatch(FracNoiseErr);
  }
  if (GBIsSameAs(gb, gbRef) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoiseGet3D NOK");
    PBErrCatch(FracNoiseErr);
  }
  VecFloat2D p2 = VecFloatCreateStatic2D();
  do {
    for (int i = 2; i--;)
      VecSet(&p2, i, (float)VecGet(&pos, i) / 64.0);
    float pn = PerlinNoiseGet(noise, &p2);
    unsigned char n = (unsigned char)floor(255.0 * pn);
    pix._rgba[GBPixelRed] = n;
    pix._rgba[GBPixelGreen] = n;
```

```
    pix._rgba[GBPixelBlue] = n;
    GBSetFinalPixel(gb, &pos, &pix);
  } while (VecStep(&pos, &dim));
  // Compare to reference
  GBFree(&gbRef);
  gbRef = GBCreateFromFile("./UnitTestPerlinNoise2DRef.tga");
  if (gbRef == NULL) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg,
      "Couln't load UnitTestPerlinNoise2DRef.tga");
    PBErrCatch(FracNoiseErr);
  }
  if (GBIsSameAs(gb, gbRef) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoiseGet2D NOK");
    PBErrCatch(FracNoiseErr);
  }
  float p1 = 0.0;
  int i = 0;
  float check[64] = {
    0.500000,0.507794,0.515479,0.522958,0.530141,0.536949,0.543313,
    0.549172,0.554474,0.559176,0.563242,0.566646,0.569366,0.571388,
    0.572708,0.573324,0.573242,0.572474,0.571037,0.568952,0.566247,
    0.562952,0.559102,0.554736,0.549896,0.544627,0.538977,0.532996,
    0.526736,0.520251,0.513596,0.506826,0.500000,0.493174,0.486404,
    0.479749,0.473264,0.467004,0.461023,0.455373,0.450104,0.445264,
    0.440898,0.437048,0.433753,0.431048,0.428963,0.427526,0.426758,
    0.426676,0.427292,0.428612,0.430634,0.433354,0.436758,0.440824,
    0.445526,0.450828,0.456687,0.463051,0.469859,0.477042,0.484521,
    0.492206};
  do {
    float pn = PerlinNoiseGet(noise, p1);
    if (ISEQUALF(pn, check[i]) == false) {
      FracNoiseErr->_type = PBErrTypeUnitTestFailed;
      sprintf(FracNoiseErr->_msg, "PerlinNoiseGet1D NOK");
      PBErrCatch(FracNoiseErr);
    }
    p1 += 1.0 / 64.0;
    ++i;
  } while (p1 < 1.0);
  int permut[256] = {};
  for (int i = 256; i--;)
    permut[i] = i;
  PerlinNoiseSetPermut(noise, permut);
  for (int i = 256; i--;)
    if (noise->_p[i] != permut[i] ||
      noise->_p[256 + i] != permut[i]) {
      FracNoiseErr->_type = PBErrTypeUnitTestFailed;
      sprintf(FracNoiseErr->_msg, "PerlinNoiseSetPermut NOK");
      PBErrCatch(FracNoiseErr);
    }
  // Free memory
  GBFree(&gb);
  GBFree(&gbRef);
  PerlinNoiseFree(&noise);
  printf("UnitTestPerlinNoise OK\n");
}

void UnitTestPerlinNoisePodCreateFree() {
  VecFloat3D seed = VecFloatCreateStatic3D();
  VecSet(&seed, 0, 1.0); VecSet(&seed, 1, 2.0); VecSet(&seed, 2, 3.0);
  VecShort2D dim = VecShortCreateStatic2D();
```

```
  VecSet(&dim, 0, 3); VecSet(&dim, 1, 2);
  PerlinNoisePod* pod = PerlinNoisePodCreate(&dim, &seed);
  if (pod->_bound != NULL ||
    ISEQUALF(pod->_border, 0.1) == false ||
    ISEQUALF(pod->_smooth, 1.0) == false ||
    ISEQUALF(pod->_square, 0.0) == false ||
    ISEQUALF(VecGet(pod->_scaleIn, 0), 1.0) == false ||
    ISEQUALF(VecGet(pod->_scaleIn, 1), 1.0) == false ||
    ISEQUALF(VecGet(pod->_scaleIn, 2), 1.0) == false ||
    VecGetDim(pod->_scaleOut) != 2 ||
    ISEQUALF(VecGet(pod->_scaleOut, 0), 1.0) == false ||
    ISEQUALF(VecGet(pod->_scaleOut, 1), 1.0) == false ||
    pod->_fractalLvl != 0 ||
    ISEQUALF(pod->_fractalCoeff, 0.1) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodCreate NOK");
    PBErrCatch(FracNoiseErr);
  }
  VecNormalise(&seed);
  if (VecIsEqual(&(pod->_seed), &seed) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodCreate NOK");
    PBErrCatch(FracNoiseErr);
  }
  PerlinNoisePodFree(&pod);
  if (pod != NULL) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodFree NOK");
    PBErrCatch(FracNoiseErr);
  }
  printf("UnitTestPerlinNoisePodCreateFree OK\n");
}

void UnitTestPerlinNoisePodSetGet() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
  PerlinNoisePod* pod = PerlinNoisePodCreate(&dim, NULL);
  if (PerlinNoisePodNoise(pod) != &(pod->_noise)) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodNoise NOK");
    PBErrCatch(FracNoiseErr);
  }
  if (PerlinNoisePodSeed(pod) != &(pod->_seed)) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodSeed NOK");
    PBErrCatch(FracNoiseErr);
  }
  VecFloat3D seed = VecFloatCreateStatic3D();
  VecSet(&seed, 0, 1.0); VecSet(&seed, 1, 2.0); VecSet(&seed, 2, 3.0);
  PerlinNoisePodSetSeed(pod, &seed);
  VecNormalise(&seed);
  if (VecIsEqual(PerlinNoisePodSeed(pod), &seed) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetSeed NOK");
    PBErrCatch(FracNoiseErr);
  }
  if (PerlinNoisePodBound(pod) != pod->_bound) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodBound NOK");
    PBErrCatch(FracNoiseErr);
  }
  Facoid* shap = FacoidCreate(2);
```

```
PerlinNoisePodSetBound(pod, shap);
if (ShapoidIsEqual(PerlinNoisePodBound(pod), shap) == false) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetBound NOK");
  PBErrCatch(FracNoiseErr);
}
ShapoidFree(&shap);
if (PerlinNoisePodScaleIn(pod) != pod->_scaleIn) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodScaleIn NOK");
  PBErrCatch(FracNoiseErr);
}
VecFloat2D in = VecFloatCreateStatic2D();
VecSet(&in, 0, 2.0); VecSet(&in, 1, 3.0);
PerlinNoisePodSetScaleIn(pod, (VecFloat*)&in);
if (VecIsEqual(PerlinNoisePodScaleIn(pod), &in) == false) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetScaleIn NOK");
  PBErrCatch(FracNoiseErr);
}
if (PerlinNoisePodScaleOut(pod) != pod->_scaleOut) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodScaleOut NOK");
  PBErrCatch(FracNoiseErr);
}
VecFloat3D out = VecFloatCreateStatic3D();
VecSet(&out, 0, 2.0); VecSet(&out, 1, 3.0); VecSet(&out, 2, 4.0);
PerlinNoisePodSetScaleOut(pod, (VecFloat*)&out);
if (VecIsEqual(PerlinNoisePodScaleOut(pod), &out) == false) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetScaleOut NOK");
  PBErrCatch(FracNoiseErr);
}
if (PerlinNoisePodShiftOut(pod) != pod->_shiftOut) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodShiftOut NOK");
  PBErrCatch(FracNoiseErr);
}
VecFloat3D shift = VecFloatCreateStatic3D();
VecSet(&shift, 0, 2.0); VecSet(&shift, 1, 3.0); VecSet(&shift, 2, 4.0);
PerlinNoisePodSetShiftOut(pod, (VecFloat*)&shift);
if (VecIsEqual(PerlinNoisePodShiftOut(pod), &shift) == false) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetShiftOut NOK");
  PBErrCatch(FracNoiseErr);
}
if (PerlinNoisePodGetFractLvl(pod) != pod->_fractalLvl) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodGetFractLvl NOK");
  PBErrCatch(FracNoiseErr);
}
PerlinNoisePodSetFractLvl(pod, 2);
if (PerlinNoisePodGetFractLvl(pod) != 2) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetFractLvl NOK");
  PBErrCatch(FracNoiseErr);
}
if (ISEQUALF(PerlinNoisePodGetFractCoeff(pod), pod->_fractalCoeff) ==
  false) {
  FracNoiseErr->_type = PBErrTypeUnitTestFailed;
  sprintf(FracNoiseErr->_msg, "PerlinNoisePodGetFractCoeff NOK");
  PBErrCatch(FracNoiseErr);
```

```
  }
  PerlinNoisePodSetFractCoeff(pod, 0.5);
  if (ISEQUALF(PerlinNoisePodGetFractCoeff(pod), 0.5) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetFractCoeff NOK");
    PBErrCatch(FracNoiseErr);
  }
  if (ISEQUALF(PerlinNoisePodGetBorder(pod), pod->_border) ==
    false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodGetBorder NOK");
    PBErrCatch(FracNoiseErr);
  }
  PerlinNoisePodSetBorder(pod, 0.25);
  if (ISEQUALF(PerlinNoisePodGetBorder(pod), 0.25) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetBorder NOK");
    PBErrCatch(FracNoiseErr);
  }
  if (ISEQUALF(PerlinNoisePodGetSmooth(pod), pod->_smooth) ==
    false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodGetSmooth NOK");
    PBErrCatch(FracNoiseErr);
  }
  PerlinNoisePodSetSmooth(pod, 0.25);
  if (ISEQUALF(PerlinNoisePodGetSmooth(pod), 0.25) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetSmooth NOK");
    PBErrCatch(FracNoiseErr);
  }
  if (ISEQUALF(PerlinNoisePodGetSquare(pod), pod->_square) ==
    false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodGetSquare NOK");
    PBErrCatch(FracNoiseErr);
  }
  PerlinNoisePodSetSquare(pod, 0.1);
  if (ISEQUALF(PerlinNoisePodGetSquare(pod), 0.1) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "PerlinNoisePodSetSquare NOK");
    PBErrCatch(FracNoiseErr);
  }
  PerlinNoisePodFree(&pod);
  printf("UnitTestPerlinNoisePodSetGet OK\n");
}

void UnitTestPerlinNoisePodGetInsideness() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
  PerlinNoisePod* pod = PerlinNoisePodCreate(&dim, NULL);
  Facoid* shap = FacoidCreate(2);
  PerlinNoisePodSetBound(pod, shap);
  PerlinNoisePodSetBorder(pod, 0.5);
  VecFloat2D pos = VecFloatCreateStatic2D();
  float check[25] = {
    0.000000,0.000000,0.000000,0.003368,0.113456,0.537834,0.955768,
    1.000000,1.000000,1.000000,1.000000,1.000000,1.000000,1.000000,
    1.000000,1.000000,1.000000,1.000000,0.955768,0.537834,0.113456,
    0.003368,0.000000,0.000000,0.000000};
  for (int i = 0; i <= 24; ++i) {
    float x = -0.1 + 0.05 * (float)i;
```

```
    VecSet(&pos, 0, x); VecSet(&pos, 1, x);
    float in = PerlinNoisePodGetInsideness(pod, (VecFloat*)&pos);
    if (ISEQUALF(in, check[i]) == false) {
      FracNoiseErr->_type = PBErrTypeUnitTestFailed;
      sprintf(FracNoiseErr->_msg, "PerlinNoisePodGetInsideness NOK");
      PBErrCatch(FracNoiseErr);
    }
  }
  ShapoidFree(&shap);
  PerlinNoisePodFree(&pod);
  printf("UnitTestPerlinNoisePodGetInsideness OK\n");
}

void UnitTestPerlinNoisePod() {
  UnitTestPerlinNoisePodCreateFree();
  UnitTestPerlinNoisePodSetGet();
  UnitTestPerlinNoisePodGetInsideness();

  printf("UnitTestPerlinNoisePod OK\n");
}

void UnitTestFracNoiseCreateFree() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
  FracNoise* noise = FracNoiseCreate(&dim, NULL);
  if (VecIsEqual(&(noise->_dim), &dim) != true) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseCreate NOK");
    PBErrCatch(FracNoiseErr);
  }
  FracNoiseFree(&noise);
  if (noise != NULL) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseFree NOK");
    PBErrCatch(FracNoiseErr);
  }
  printf("UnitTestFracNoiseCreateFree OK\n");
}

void UnitTestFracNoiseSetGet() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
  FracNoise* noise = FracNoiseCreate(&dim, NULL);
  if (VecIsEqual(FracNoiseDim(noise), &dim) != true) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseDim NOK");
    PBErrCatch(FracNoiseErr);
  }
  if (FracNoisePods(noise) != &(noise->_noises)) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoisePods NOK");
    PBErrCatch(FracNoiseErr);
  }
  if (FracNoiseGetNoise(noise, 0) !=
    (PerlinNoisePod*)(noise->_noises._head->_data)) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseGetNoise NOK");
    PBErrCatch(FracNoiseErr);
  }
  FracNoiseFree(&noise);
  printf("UnitTestFracNoiseSetGet OK\n");
}
```

```
void UnitTestFracNoiseAddRemoveNoise() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
  FracNoise* noise = FracNoiseCreate(&dim, NULL);
  PerlinNoisePod* pod = FracNoiseAddNoise(noise, NULL);
  if (GSetNbElem(FracNoisePods(noise)) != 2 ||
    pod != FracNoiseGetNoise(noise, 1)) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseAddNoise NOK");
    PBErrCatch(FracNoiseErr);
  }
  FracNoiseRemoveNoise(noise, pod);
  if (GSetNbElem(FracNoisePods(noise)) != 1 ||
    pod == FracNoiseGetNoise(noise, 0)) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseRemoveNoise NOK");
    PBErrCatch(FracNoiseErr);
  }
  FracNoiseFree(&noise);
  printf("UnitTestFracNoiseAddRemoveNoise OK\n");
}

void UnitTestFracNoiseGet() {
  // Create a FracNoise
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
  FracNoise* noise = FracNoiseCreate(&dim, NULL);
  // Set properties of the FracNoise
  PerlinNoisePod* podA = FracNoiseGetNoise(noise, 0);
  VecFloat2D scaleIn = VecFloatCreateStatic2D();
  VecSet(&scaleIn, 0, 1.0); VecSet(&scaleIn, 1, 0.5);
  PerlinNoisePodSetScaleIn(podA, (VecFloat*)&scaleIn);
  VecFloat3D scaleOut = VecFloatCreateStatic3D();
  VecSet(&scaleOut, 0, 1.0); VecSet(&scaleOut, 1, 0.2);
  VecSet(&scaleOut, 2, 1.0);
  PerlinNoisePodSetScaleOut(podA, (VecFloat*)&scaleOut);
  PerlinNoisePodSetFractLvl(podA, 1);
  VecFloat3D seed = VecFloatCreateStatic3D();
  VecSet(&seed, 0, -1.0); VecSet(&seed, 1, 0.5);
  VecSet(&seed, 2, 1.0);
  PerlinNoisePod* podB = FracNoiseAddNoise(noise, &seed);
  VecSet(&scaleOut, 0, 0.0); VecSet(&scaleOut, 1, 0.7);
  VecSet(&scaleOut, 2, 0.0);
  PerlinNoisePodSetScaleOut(podB, (VecFloat*)&scaleOut);
  VecFloat3D shiftOut = VecFloatCreateStatic3D();
  VecSet(&shiftOut, 0, 0.0); VecSet(&shiftOut, 1, 0.1);
  VecSet(&shiftOut, 2, 0.0);
  Spheroid* bound = SpheroidCreate(2);
  ShapoidScale(bound, (float)(200.0 / 64.0));
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 250.0 / 64.0); VecSet(&v, 1, 250.0 / 64.0);
  ShapoidTranslate(bound, &v);
  PerlinNoisePodSetBound(podB, bound);
  PerlinNoisePodSetBorder(podB, 0.5);
  PerlinNoisePodSetSmooth(podA, 3.0);
  // Create a GB to render the image
  VecSet(&dim, 0, 512); VecSet(&dim, 1, 512);
  GenBrush* gb = GBCreateImage(&dim);
  // Draw the Image
  VecShort2D pos = VecShortCreateStatic2D();
  VecFloat2D p = VecFloatCreateStatic2D();
```

```
  GBPixel pix = GBColorBlack;
  do {
    for (int i = 2; i--;)
      VecSet(&p, i, (float)VecGet(&pos, i) / 64.0);
    VecFloat* pn = FracNoiseGet(noise, &p);
    pix._rgba[GBPixelRed] = (unsigned char)floor(255.0 * VecGet(pn, 0));
    pix._rgba[GBPixelGreen] =
      (unsigned char)floor(255.0 * VecGet(pn, 1));
    pix._rgba[GBPixelBlue] = (unsigned char)floor(255.0 * VecGet(pn, 2));
    GBSetFinalPixel(gb, &pos, &pix);
    VecFree(&pn);
  } while (VecStep(&pos, &dim));
  // Compare to reference (two versions because changes in compilation
  // options trigger small variations when casting from float to int)
#if BUILDMODE == 0
  GenBrush* gbRef = GBCreateFromFile("./UnitTestFracNoiseGetRef0.tga");
#else
  GenBrush* gbRef = GBCreateFromFile("./UnitTestFracNoiseGetRef1.tga");
#endif
  if (gbRef == NULL) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg,
      "Couln't load UnitTestPerlinNoise3DRef.tga");
    PBErrCatch(FracNoiseErr);
  }
  if (GBIsSameAs(gb, gbRef) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseGet NOK");
    PBErrCatch(FracNoiseErr);
  }
  // Free memory
  ShapoidFree(&bound);
  GBFree(&gb);
  GBFree(&gbRef);
  FracNoiseFree(&noise);

  printf("UnitTestFracNoiseGet OK\n");
}

void UnitTestFracNoiseExport() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3); VecSet(&dim, 1, 1);
  FracNoise* noise = FracNoiseCreate(&dim, NULL);
  VecFloat3D range = VecFloatCreateStatic3D();
  VecSet(&range, 0, 10.0); VecSet(&range, 1, 10.0);
  VecSet(&range, 2, 10.0);
  VecShort3D nbSample = VecShortCreateStatic3D();
  VecSet(&nbSample, 0, 30); VecSet(&nbSample, 1, 30);
  VecSet(&nbSample, 2, 30);
  int res = 8;
  char* fileName = "./UnitTestFracNoiseExport.df3";
  if (FracNoiseExportDF3(noise, &range, &nbSample, res, true,
    fileName) == false) {
    FracNoiseErr->_type = PBErrTypeUnitTestFailed;
    sprintf(FracNoiseErr->_msg, "FracNoiseExportDF3 NOK");
    PBErrCatch(FracNoiseErr);
  }
  FracNoiseFree(&noise);
  printf("UnitTestFracNoiseExport OK\n");
}

void UnitTestFracNoise() {
```

```
  UnitTestFracNoiseCreateFree();
  UnitTestFracNoiseSetGet();
  UnitTestFracNoiseAddRemoveNoise();
  UnitTestFracNoiseGet();
  UnitTestFracNoiseExport();

  printf("UnitTestFracNoise OK\n");
}

void UnitTestAll() {
  UnitTestPerlinNoise();
  UnitTestPerlinNoisePod();
  UnitTestFracNoise();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();

  // Return success code
  return 0;
}
```

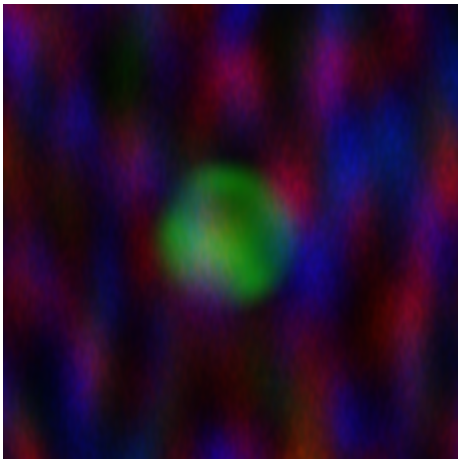
6 Unit tests output

```
UnitTestPerlinNoise OK
UnitTestPerlinNoisePodCreateFree OK
UnitTestPerlinNoisePodSetGet OK
UnitTestPerlinNoisePodGetInsideness OK
UnitTestPerlinNoisePod OK
UnitTestFracNoiseCreateFree OK
UnitTestFracNoiseSetGet OK
UnitTestFracNoiseAddRemoveNoise OK
UnitTestFracNoiseGet OK
UnitTestFracNoiseExport OK
UnitTestFracNoise OK
UnitTestAll OK
```

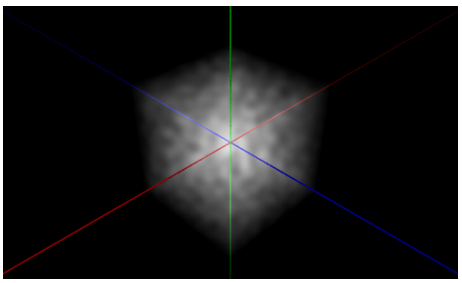
UnitTestPerlinNoise2DRef.png:

UnitTestPerlinNoise3DRef.png:

UnitTestFracNoiseGetRef.png:



UnitTestFracNoiseExportRef.df3 (rendered in POV-Ray in a unit cube):



7 Examples

7.1 Terrain

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "fracnoise.h"
#include "genbrush.h"

#define RANDOMSEED 1

int main() {
```

```
srandom(RANDOMSEED);
// HF dimension
int HFDim = 512;
// Create a FracNoise
VecShort2D dim = VecShortCreateStatic2D();
VecSet(&dim, 0, 2); VecSet(&dim, 1, 1);
VecFloat3D seed = VecFloatCreateStatic3D();
for (int i = 3; i--;)
  VecSet(&seed, i, rnd());
FracNoise* terrain = FracNoiseCreate(&dim, &seed);
// --- main ground
PerlinNoisePod* ground = FracNoiseGetNoise(terrain, 0);
// scaleIn
VecFloat2D scaleIn = VecFloatCreateStatic2D();
float sIn = 1.0 / (float)HFDim;
for (int i = 2; i--;)
  VecSet(&scaleIn, i, sIn);
PerlinNoisePodSetScaleIn(ground, &scaleIn);
// scaleOut
float sOut = 200.0;
VecFloat* scaleOut = VecFloatCreate(1);
VecSet(scaleOut, 0, sOut);
PerlinNoisePodSetScaleOut(ground, scaleOut);
// fractal
PerlinNoisePodSetFractLvl(ground, 2);
PerlinNoisePodSetFractCoeff(ground, 0.3);
// --- mountains
PerlinNoisePod* mount = FracNoiseAddNoise(terrain, &seed);
// scaleIn
sIn = 5.0 / (float)HFDim;
for (int i = 2; i--;)
  VecSet(&scaleIn, i, sIn);
PerlinNoisePodSetScaleIn(mount, &scaleIn);
// scaleOut
sOut = 500.0;
VecSet(scaleOut, 0, sOut);
PerlinNoisePodSetScaleOut(mount, scaleOut);
// boundary
Spheroid* boundMount = SpheroidCreate(2);
PerlinNoisePodSetBound(mount, boundMount);
VecFloat2D scaleBound = VecFloatCreateStatic2D();
VecSet(&scaleBound, 0, 350.0); VecSet(&scaleBound, 1, 750.0);
ShapoidScale(boundMount, (VecFloat*)&scaleBound);
ShapoidRotCenter(boundMount, PBMATH_QUARTERPI * 0.5);
VecFloat2D shiftBound = VecFloatCreateStatic2D();
VecSet(&shiftBound, 0, 150.0); VecSet(&shiftBound, 1, 275.0);
ShapoidTranslate(boundMount, &shiftBound);
PerlinNoisePodSetBorder(mount, 1.0);
// fractal
PerlinNoisePodSetFractLvl(mount, 2);
PerlinNoisePodSetFractCoeff(mount, 0.4);
// Create a GB to render the height field
VecSet(&dim, 0, HFDim); VecSet(&dim, 1, HFDim);
GenBrush* gb = GBCreateImage(&dim);
// Draw the image
VecShort2D pos = VecShortCreateStatic2D();
VecFloat2D p = VecFloatCreateStatic2D();
GBPixel pix = GBColorBlack;
// search the min and max values ot renormalize over the height field
float min = 255.0;
float max = 0.0;
do {
```

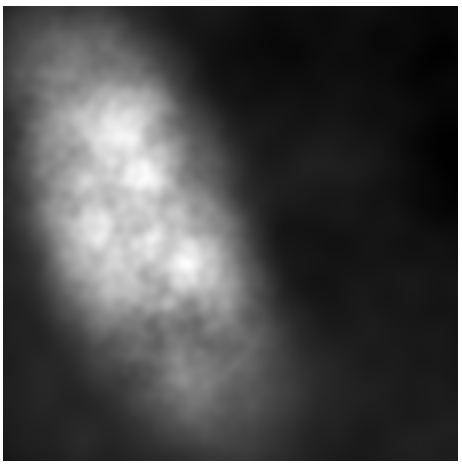
```
    for (int i = 2; i--;)
      VecSet(&p, i, (float)VecGet(&pos, i));
    VecFloat* pn = FracNoiseGet(terrain, &p);
    if (VecGet(pn, 0) > max)
      max = VecGet(pn, 0);
    if (VecGet(pn, 0) < min)
      min = VecGet(pn, 0);
    VecFree(&pn);
  } while (VecStep(&pos, &dim));
  VecSetNull(&pos);
  do {
    for (int i = 2; i--;)
      VecSet(&p, i, (float)VecGet(&pos, i));
    VecFloat* pn = FracNoiseGet(terrain, &p);
    unsigned char rgb = (unsigned char)floor(255.0 *
      (VecGet(pn, 0) - min) / (max - min));
    // Avoid hole in the height field due to water_level in POV
    if (rgb == 0) rgb = 1;
    pix._rgba[GBPixelRed] = rgb;
    pix._rgba[GBPixelGreen] = rgb;
    pix._rgba[GBPixelBlue] = rgb;
    GBSetFinalPixel(gb, &pos, &pix);
    VecFree(&pn);
  } while (VecStep(&pos, &dim));
  // Save the image
  GBSetFileName(gb, "./HF.tga");
  GBRender(gb);
  // Free memory
  GBFree(&gb);
  FracNoiseFree(&terrain);
  VecFree(&scaleOut);
  ShapoidFree(&boundMount);

  // Return success code
  return 0;
}
```

terrain.pov:

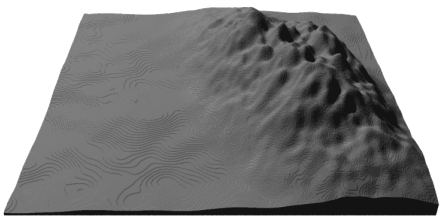
```
#include "colors.inc"

#declare _texNeutral = texture {
  pigment { color rgb <0.75, 0.75, 0.75> }
  finish { ambient 0.1 diffuse 0.6 phong 0.0}
}

#declare _tex = texture {
  pigment { color White }
}

#declare RndSeed = seed(30);
#declare _posCamera = <0.0,1.0,2.0>;
#declare _lookAt = <0.0,0.0,0.5>;

camera {
  location    _posCamera
  look_at     _lookAt
  right x
  up y * 3/5
}
```

```
light_source {
  <1.0, 2.0, 0.0>
  color rgb 1.0
  area_light <-0.1, 0, -0.1>, <0.1, 0, 0.1>, 3, 3
  adaptive 1
  jitter
}

background { color rgbft <1.0, 1.0, 1.0, 1.0, 1.0> }

intersection {
  height_field {
    tga
    "HF.tga"
  }
  box {
    <0.0001,0.0001,0.0001>
    <0.9999,1.0,0.9999>
  }
  translate -0.5 * x
  scale <1.0, 0.25, 1.0>
  texture { _texNeutral }
}
```

HF.png:



terrain.png:



7.2 Cloud

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "fracnoise.h"
#include "genbrush.h"

#define RANDOMSEED 2

// I have no idea if the names match, you'd be warned !!
typedef enum {
  cumulusA, cumulusB, stratusA, stratusB, stratusC
} typeCloud;

void MakeCloud(typeCloud type, int rndSeed, char* fileName) {
  srandom(rndSeed);
  //srandom(time(NULL));
  // Set parameters according to type
  float smoothness = 1.0;
  float nbPodMin = 50.0;
  float nbPodMax = 100.0;
  float sizeCompMin = 0.1;
  float sizeCompMax = 0.3;
  float scaleInY = 1.0;
  float borderMin = 0.1;
  float borderMax = 0.5;
  if (type == cumulusA) {
    smoothness = 0.1;
    nbPodMin = 50.0;
    nbPodMax = 100.0;
    sizeCompMin = 0.1;
    sizeCompMax = 0.3;
    scaleInY = 1.0;
  } else if (type == cumulusB) {
    smoothness = 0.1;
    nbPodMin = 100.0;
    nbPodMax = 150.0;
    sizeCompMin = 0.1;
    sizeCompMax = 0.3;
    scaleInY = 1.0;
    borderMin = 0.0;
    borderMax = 0.0;
  } else if (type == stratusA) {
```

```
  smoothness = 1.0;
  nbPodMin = 100.0;
  nbPodMax = 200.0;
  sizeCompMin = 0.05;
  sizeCompMax = 0.15;
  scaleInY = 1.0;
} else if (type == stratusB) {
  smoothness = 0.1;
  nbPodMin = 100.0;
  nbPodMax = 200.0;
  sizeCompMin = 0.05;
  sizeCompMax = 0.15;
  scaleInY = 1.0;
} else if (type == stratusC) {
  smoothness = 1.0;
  nbPodMin = 20.0;
  nbPodMax = 50.0;
  sizeCompMin = 0.1;
  sizeCompMax = 0.8;
  scaleInY = 0.2;
}
// Create the noise
VecShort2D dim = VecShortCreateStatic2D();
VecSet(&dim, 0, 3); VecSet(&dim, 1, 1);
FracNoise* cloud = FracNoiseCreate(&dim, NULL);
// Create the cloud components
int nbPod = (int)floor(nbPodMin + rnd() * (nbPodMax- nbPodMin));
VecFloat3D seed = VecFloatCreateStatic3D();
Spheroid** bounds = malloc(nbPod * sizeof(Spheroid*));
VecFloat3D scaleIn = VecFloatCreateStatic3D();
VecFloat3D scaleBound = VecFloatCreateStatic3D();
PerlinNoisePod* pod = NULL;
for (int iPod = nbPod; iPod--;) {
  if (iPod == 0)
    pod = FracNoiseGetNoise(cloud, 0);
  else {
    for (int i = 3; i--;)
      VecSet(&seed, i, rnd());
    pod = FracNoiseAddNoise(cloud, &seed);
  }
  bounds[iPod] = SpheroidCreate(3);
  PerlinNoisePodSetBound(pod, bounds[iPod]);
  float border = borderMin + (borderMax - borderMin) * rnd();
  PerlinNoisePodSetBorder(pod, border);
  float s = sizeCompMin + (sizeCompMax - sizeCompMin) * rnd();
  for (int i = 3; i--;)
    VecSet(&scaleBound, i, s * (i == 1 ? scaleInY : 1.0));
  ShapoidScale(bounds[iPod], (VecFloat*)&scaleBound);
  for (int i = 3; i--;) {
    float x = s + (1.0 - 2.0 * s) * rnd();
    if (type != cumulusB && i == 1 && x < 0.5)
      x = 0.5 - pow(0.5 - x, 2.0);
    VecSet(ShapoidPos(bounds[iPod]), i, x);
  }
  float sIn = 20.0 + 40.0 * rnd();
  for (int i = 3; i--;)
    VecSet(&scaleIn, i, sIn);
  PerlinNoisePodSetScaleIn(pod, &scaleIn);
  PerlinNoisePodSetSmooth(pod, smoothness);
}
// Export to the df3 file
VecFloat3D range = VecFloatCreateStatic3D();
```

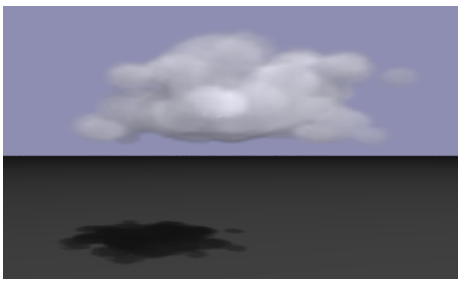
```
  VecSet(&range, 0, 1.0); VecSet(&range, 1, 1.0);
  VecSet(&range, 2, 1.0);
  VecShort3D nbSample = VecShortCreateStatic3D();
  VecSet(&nbSample, 0, 50); VecSet(&nbSample, 1, 50);
  VecSet(&nbSample, 2, 50);
  int res = 16;
  if (FracNoiseExportDF3(cloud, &range, &nbSample, res, false,
    fileName) == false) {
    fprintf(stderr, "export to df3 failed\n");
  }
  // Free memory
  FracNoiseFree(&cloud);
  for (int iPod = nbPod; iPod--;)
    ShapoidFree(bounds + iPod);
  free(bounds);

}

int main(int argc, char** argv) {
  (void)argc;(void)argv;
  char* fileName = "./cloud.df3";
  MakeCloud(cumulusB, RANDOMSEED, fileName);
  int ret = system("make DF3");(void)ret;

  //MakeCloud((typeCloud)atoi(argv[1]), RANDOMSEED, argv[2]);


  // Return success code
  return 0;
}
```

cloud.pov:

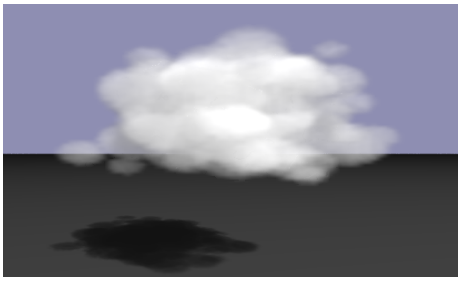
```
#include "colors.inc"

#declare _posCamera = <2.5,0.0,2.5>;
#declare _lookAt = <0.0,0.1,0.0>;

camera {
  location    _posCamera
  look_at     _lookAt
  right x
  up y * 3/5
}

light_source {
  <10.0, 10.0, 20.5>
  color rgb 1.0
}

background { color rgb <0.0, 0.0, 0.0> }

#declare boxinterior = interior {
  media {
    method 3
    samples 10,10
    emission <0.9, 0.9, 1>
    //emission <1.25, 1.25, 1.25> // cumulusB
    absorption <1.0, 1.0, 1.0>*2
    scattering {1, <1, 1, 1>*1 extinction 1.0 }
```

```
    density {
      density_file df3 "./cloud.df3"
        interpolate 1
    }
  }
}

#declare boxtexture = texture {
  pigment {
    rgbf 1
  }
}

box {
  <0,0,0>, <1,1,1>
  texture { boxtexture }
  interior { boxinterior }
  hollow
  scale <2,1,2>
  translate <-0.5,-0.25,-0.5>
}

plane {
  y, -1.0
  pigment { color rgb 0.7 }
}

sphere {
  <0.0,0.0,0.0>, 1000.0
  pigment {color <0.8,0.8,1.0>}
  hollow
}
```

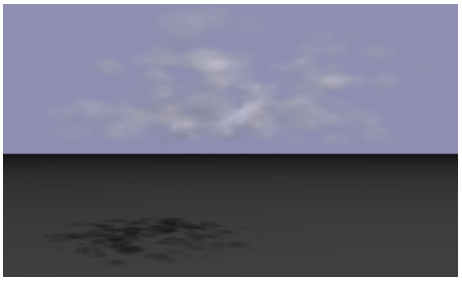
cumulusA.png:



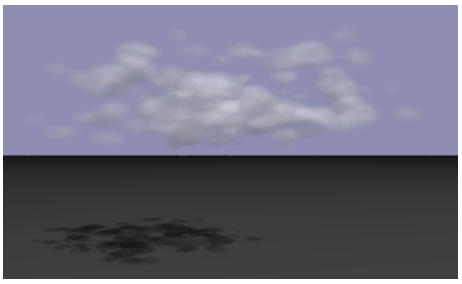
cumulusB.png:



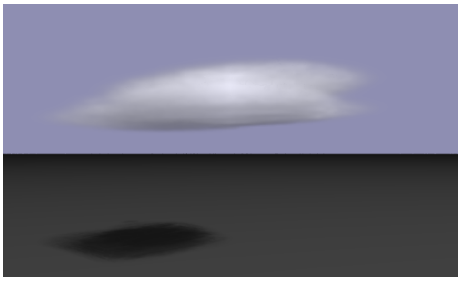
stratusA.png:



stratusB.png:



stratusC.png:



7.3 Rock

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "fracnoise.h"
#include "genbrush.h"

#define RANDOMSEED 30

typedef enum {
  rockA, rockB
} typeRock;

void MakeRockA(typeRock type, int rndSeed, char* fileName,
  VecFloat3D* radiusRock) {
  srandom(rndSeed);
  // Parameters
  float fractCoeff = 0.1;
  float smooth = 0.5;
  float scaleInXZ = 3.0;
  float scaleInY = 5.0;
  float scaleOut = 0.3;
  fractCoeff = 0.05 + 0.15 * rnd();
  smooth = 0.1 + 1.9 * rnd();
  scaleInXZ = 1.0 + 4.0 * rnd();
  scaleInY = 1.0 + 4.0 * rnd();
  scaleOut = 0.1 + 0.2 * rnd();
  // Create the FracNoise for the rock
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3); VecSet(&dim, 1, 3);
  VecFloat3D seed = VecFloatCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&seed, i, rnd());
  FracNoise* rock = FracNoiseCreate(&dim, &seed);
  PerlinNoisePod* pod = FracNoiseGetNoise(rock, 0);
  if (type == rockB)
    PerlinNoisePodSetSquare(pod, 0.8);
  VecFloat3D scaleIn = VecFloatCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&scaleIn, i, (i == 1 ? scaleInY : scaleInXZ));
  PerlinNoisePodSetScaleIn(pod, &scaleIn);
```

```
VecFloat3D sOut = VecFloatCreateStatic3D();
for (int i = 3; i--;)
  VecSet(&sOut, i, scaleOut);
PerlinNoisePodSetScaleOut(pod, &sOut);
for (int i = 3; i--;)
  VecSet(&sOut, i, -0.5 * scaleOut);
PerlinNoisePodSetShiftOut(pod, &sOut);
if (type != rockB)
  PerlinNoisePodSetFractLvl(pod, 1);
PerlinNoisePodSetFractCoeff(pod, fractCoeff);
PerlinNoisePodSetSmooth(pod, smooth);
// Open the file to save the rock mesh
FILE* fileRock = fopen(fileName, "w");
// Print the head of the mesh
fprintf(fileRock, "#declare rock = mesh {\n");
// Loop on the polar coordinates
float theta = 0.0;
float phi = 0.0;
float delta = PBMATH_TWOPI_DIV_360 * 1.0;
VecFloat3D pos = VecFloatCreateStatic3D();
float vtheta[4];
float vphi[4];
bool flagRow = true;
for (phi = 0.0; phi < PBMATH_TWOPI; phi += delta) {
  for (theta = 0.0; theta < PBMATH_PI; theta += delta) {
    // Calculate the polar coordinates of vertices
    if (flagRow) {
      vtheta[0] = theta - delta * 0.5;
      vtheta[1] = theta;
      vtheta[2] = theta + delta * 0.5;
      vtheta[3] = theta + delta;
    } else {
      vtheta[0] = theta;
      vtheta[1] = theta - delta * 0.5;
      vtheta[2] = theta + delta;
      vtheta[3] = theta + delta * 0.5;
    }
    vphi[0] = phi + delta;
    vphi[1] = phi;
    vphi[2] = phi + delta;
    vphi[3] = phi;
    // Calculate the coordinates of vertices and write the 2 triangles
    fprintf(fileRock, "triangle {");
    for (int iv = 3; iv--;) {
      VecSet(&pos, 0,
        VecGet(radiusRock, 0) * sin(vtheta[iv]) * cos(vphi[iv]));
      VecSet(&pos, 1,
        VecGet(radiusRock, 1) * cos(vtheta[iv]));
      VecSet(&pos, 2,
        VecGet(radiusRock, 2) * sin(vtheta[iv]) * sin(vphi[iv]));
      VecFloat* vert = FracNoiseGet(rock, &pos);
      VecOp(vert, 1.0, (VecFloat*)&pos, 1.0);
      VecFloatPrint(vert, fileRock, 5);
      if (iv != 0)
        fprintf(fileRock, ",");
      VecFree(&vert);
    }
    fprintf(fileRock, "}\n");
    fprintf(fileRock, "triangle {");
    for (int iv = 1; iv < 4; ++iv) {
      VecSet(&pos, 0,
        VecGet(radiusRock, 0) * sin(vtheta[iv]) * cos(vphi[iv]));
```

```
        VecSet(&pos, 1,
          VecGet(radiusRock, 1) * cos(vtheta[iv]));
        VecSet(&pos, 2,
          VecGet(radiusRock, 2) * sin(vtheta[iv]) * sin(vphi[iv]));
        VecFloat* vert = FracNoiseGet(rock, &pos);
        VecOp(vert, 1.0, (VecFloat*)&pos, 1.0);
        VecFloatPrint(vert, fileRock, 5);
        if (iv != 3)
          fprintf(fileRock, ",");
        VecFree(&vert);
      }
      fprintf(fileRock, "}\n");
    }
    flagRow = (flagRow ? false : true);
  }
  // Print the tail of the mesh
  fprintf(fileRock, "};\n");
  fclose(fileRock);
}

void MakeRockB(typeRock type, int rndSeed, char* fileName,
  VecFloat3D* radiusRock) {
  srandom(rndSeed);
  // Parameters
  float fractCoeff = 0.1;
  float smooth = 0.5;
  float scaleInXZ = 3.0;
  float scaleInY = 5.0;
  float scaleOut = 0.3;
  fractCoeff = 0.1;
  smooth = 1.0;
  scaleInXZ = 2.0 * rnd();
  scaleInY = 2.0 * rnd();
  scaleOut = 0.1 + 0.2 * rnd();
  // Create the FracNoise for the rock
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3); VecSet(&dim, 1, 3);
  VecFloat3D seed = VecFloatCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&seed, i, rnd());
  FracNoise* rock = FracNoiseCreate(&dim, &seed);
  PerlinNoisePod* pod = FracNoiseGetNoise(rock, 0);
  if (type == rockB)
    PerlinNoisePodSetSquare(pod, 0.8);
  VecFloat3D scaleIn = VecFloatCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&scaleIn, i, (i == 1 ? scaleInY : scaleInXZ));
  PerlinNoisePodSetScaleIn(pod, &scaleIn);
  VecFloat3D sOut = VecFloatCreateStatic3D();
  for (int i = 3; i--;)
    VecSet(&sOut, i, scaleOut);
  PerlinNoisePodSetScaleOut(pod, &sOut);
  for (int i = 3; i--;)
    VecSet(&sOut, i, -0.5 * scaleOut);
  PerlinNoisePodSetShiftOut(pod, &sOut);
  if (type != rockB)
    PerlinNoisePodSetFractLvl(pod, 1);
  PerlinNoisePodSetFractCoeff(pod, fractCoeff);
  PerlinNoisePodSetSmooth(pod, smooth);
  // FracNoise for the input
  //for (int i = 3; i--;)
    //VecSet(&seed, i, rnd());
```

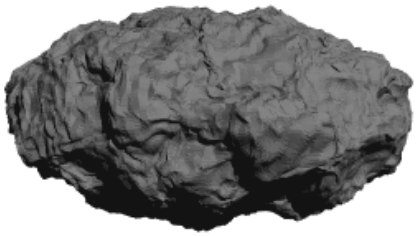
```
//FracNoise* input = FracNoiseCreate(&dim, &seed);
// Open the file to save the rock mesh
FILE* fileRock = fopen(fileName, "w");
// Print the head of the mesh
fprintf(fileRock, "#declare rock = mesh {\n");
// Loop on the polar coordinates
float theta = 0.0;
float phi = 0.0;
float delta = PBMATH_TWOPI_DIV_360 * 1.0;
VecFloat3D pos = VecFloatCreateStatic3D();
float vtheta[4];
float vphi[4];
bool flagRow = true;
for (phi = 0.0; phi < PBMATH_TWOPI; phi += delta) {
  for (theta = 0.0; theta < PBMATH_PI; theta += delta) {
    // Calculate the polar coordinates of vertices
    if (flagRow) {
      vtheta[0] = theta - delta * 0.5;
      vtheta[1] = theta;
      vtheta[2] = theta + delta * 0.5;
      vtheta[3] = theta + delta;
    } else {
      vtheta[0] = theta;
      vtheta[1] = theta - delta * 0.5;
      vtheta[2] = theta + delta;
      vtheta[3] = theta + delta * 0.5;
    }
    vphi[0] = phi + delta;
    vphi[1] = phi;
    vphi[2] = phi + delta;
    vphi[3] = phi;
    // Calculate the coordinates of vertices and write the 2 triangles
    fprintf(fileRock, "triangle {");
    for (int iv = 3; iv--;) {
      VecSet(&pos, 0,
        VecGet(radiusRock, 0) * sin(vtheta[iv]) * cos(vphi[iv]));
      VecSet(&pos, 1,
        VecGet(radiusRock, 1) * cos(vtheta[iv]));
      VecSet(&pos, 2,
        VecGet(radiusRock, 2) * sin(vtheta[iv]) * sin(vphi[iv]));
      VecFloat* vert = FracNoiseGet(rock, &pos);
      VecOp(vert, 1.0, (VecFloat*)&pos, 1.0);
      VecFloatPrint(vert, fileRock, 5);
      if (iv != 0)
        fprintf(fileRock, ",");
      VecFree(&vert);
    }
    fprintf(fileRock, "}\n");
    fprintf(fileRock, "triangle {");
    for (int iv = 1; iv < 4; ++iv) {
      VecSet(&pos, 0,
        VecGet(radiusRock, 0) * sin(vtheta[iv]) * cos(vphi[iv]));
      VecSet(&pos, 1,
        VecGet(radiusRock, 1) * cos(vtheta[iv]));
      VecSet(&pos, 2,
        VecGet(radiusRock, 2) * sin(vtheta[iv]) * sin(vphi[iv]));
      VecFloat* vert = FracNoiseGet(rock, &pos);
      VecOp(vert, 1.0, (VecFloat*)&pos, 1.0);
      VecFloatPrint(vert, fileRock, 5);
      if (iv != 3)
        fprintf(fileRock, ",");
      VecFree(&vert);
```

```
      }
      fprintf(fileRock, "}\n");
    }
    flagRow = (flagRow ? false : true);
  }
  // Print the tail of the mesh
  fprintf(fileRock, "};\n");
  fclose(fileRock);
}

void MakeRock(typeRock type, int rndSeed, char* fileName,
  VecFloat3D* radiusRock) {
  if (type == rockA)
    MakeRockA(type, rndSeed, fileName, radiusRock);
  else if (type == rockB)
    MakeRockB(type, rndSeed, fileName, radiusRock);
}

int main(int argc, char** argv) {
  (void)argc;(void)argv;
  char* fileName = "./rock.inc";

  VecFloat3D radiusRock = VecFloatCreateStatic3D();
  VecSet(&radiusRock, 0, 1.0);
  VecSet(&radiusRock, 1, 0.5);
  VecSet(&radiusRock, 2, 1.0);
  //MakeRock(rockA, RANDOMSEED, fileName, &radiusRock);
  MakeRock(rockB, time(NULL), fileName, &radiusRock);
  int ret = system("make rock");(void)ret;
  //MakeRock(rockA, time(NULL), argv[1], &radiusRock);


  // Return success code
  return 0;
}
```

rock.pov:

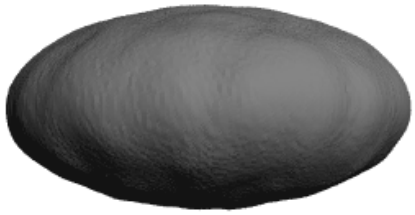
```
#include "rock.inc"
#declare _texNeutral = texture {
  pigment { color rgb <0.75, 0.75, 0.75> }
  finish { ambient 0.1 diffuse 0.6 phong 0.0}
}
camera {
  location <3.0,0.0,3.0>
  look_at <0.0,0.0,0.0>
  right x
  up y * 3/5
}
light_source {
  <10.0, 10.0, 20.5>
  color rgb 1.0
}
background { color rgb <1.0, 1.0, 1.0> }
object {
  rock
  texture {_texNeutral}
}
```

rock01.png:

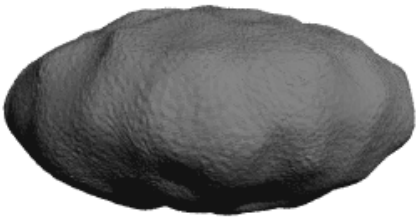


rock03.png:

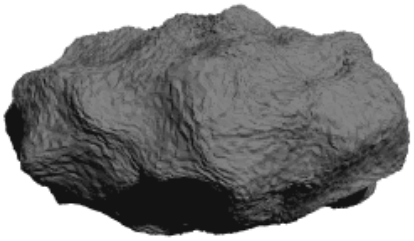
rock05.png:



rock06.png:



rock30.png:



rock50.png: