La création et la manipulation de champs de hauteur

ou l'art de changer de dimension...

Gabriel DAHAN - 28205

Travail d'Initiative Personelle Encadré (TIPE), 2025

Sommaire

Qu'est-ce qu'une carte de hauteur ?

Carte de hauteur :

- image en noir et blanc ;
- niveau de gris de chaque pixel $\Rightarrow$ hauteur dans l'espace.

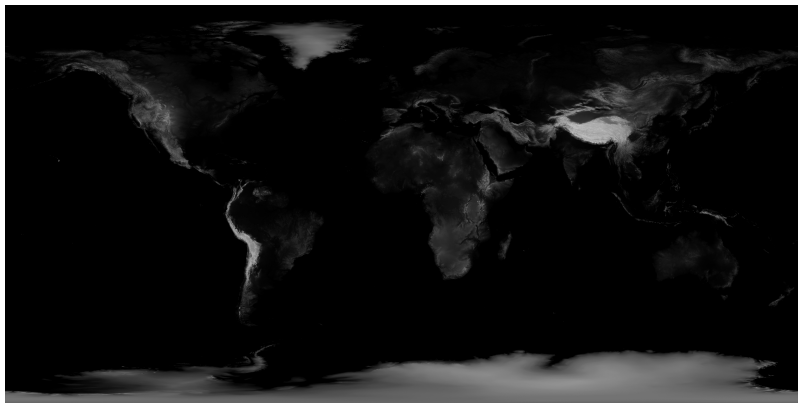


Figure: Champ de hauteur de la Terre (où le noir correspond à 0m).

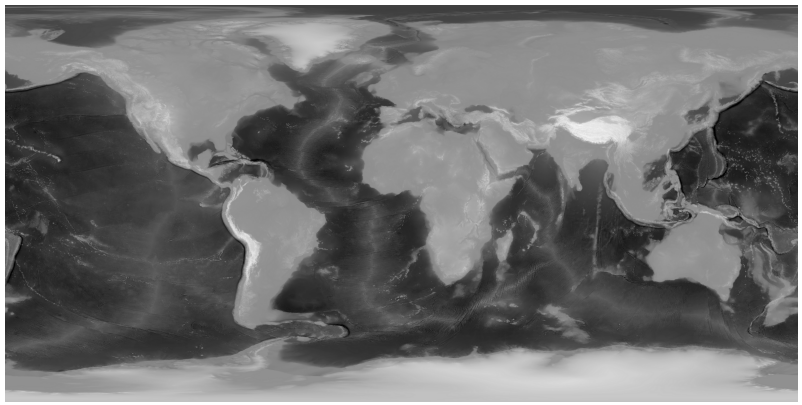


Figure: Champ de hauteur de la Terre (où le noir correspond à -11km (Mariannes)).

Caractérisation

$w, h \in \mathbb{N}^*$
$P_{w,h} = [0, w] \times [0, h]$;
$L = [\![0, 255]\!]$

Définition

Carte de hauteur H de taille $w \times h$: $H \in \mathcal{M}_{w,h}(L)$

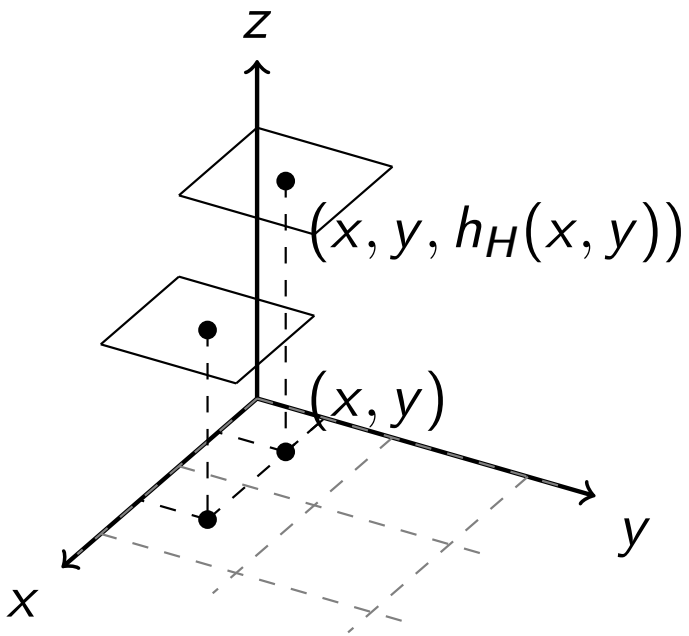
Fonction caractéristique

Fonction caractéristique de H :

$$h_H : P_{w,h} \to L$$
$$(x, y) \mapsto [H]_{\lceil x \rceil, \lceil y \rceil}$$

Du plan vers l'espace...

$$\Gamma_H : (x, y) \mapsto (x, y, h_H(x, y))$$
$$P_{w,h} \to P_{w,h} \times L$$



Rendus à l'aide de Raylib

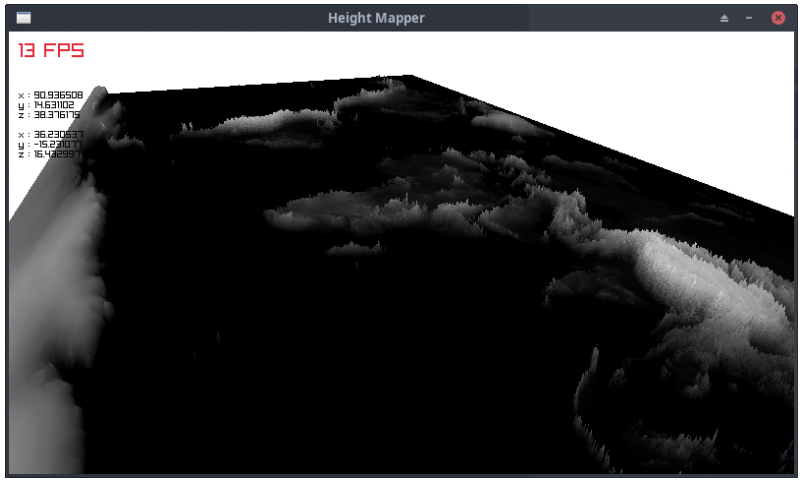


Figure: Rendu pour la figure de gauche.

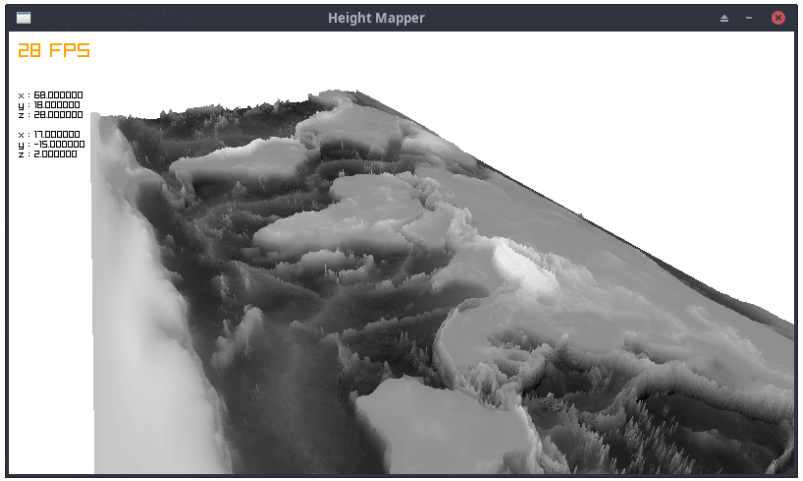


Figure: Rendu pour la figure de droite.

Ajout d'une texture

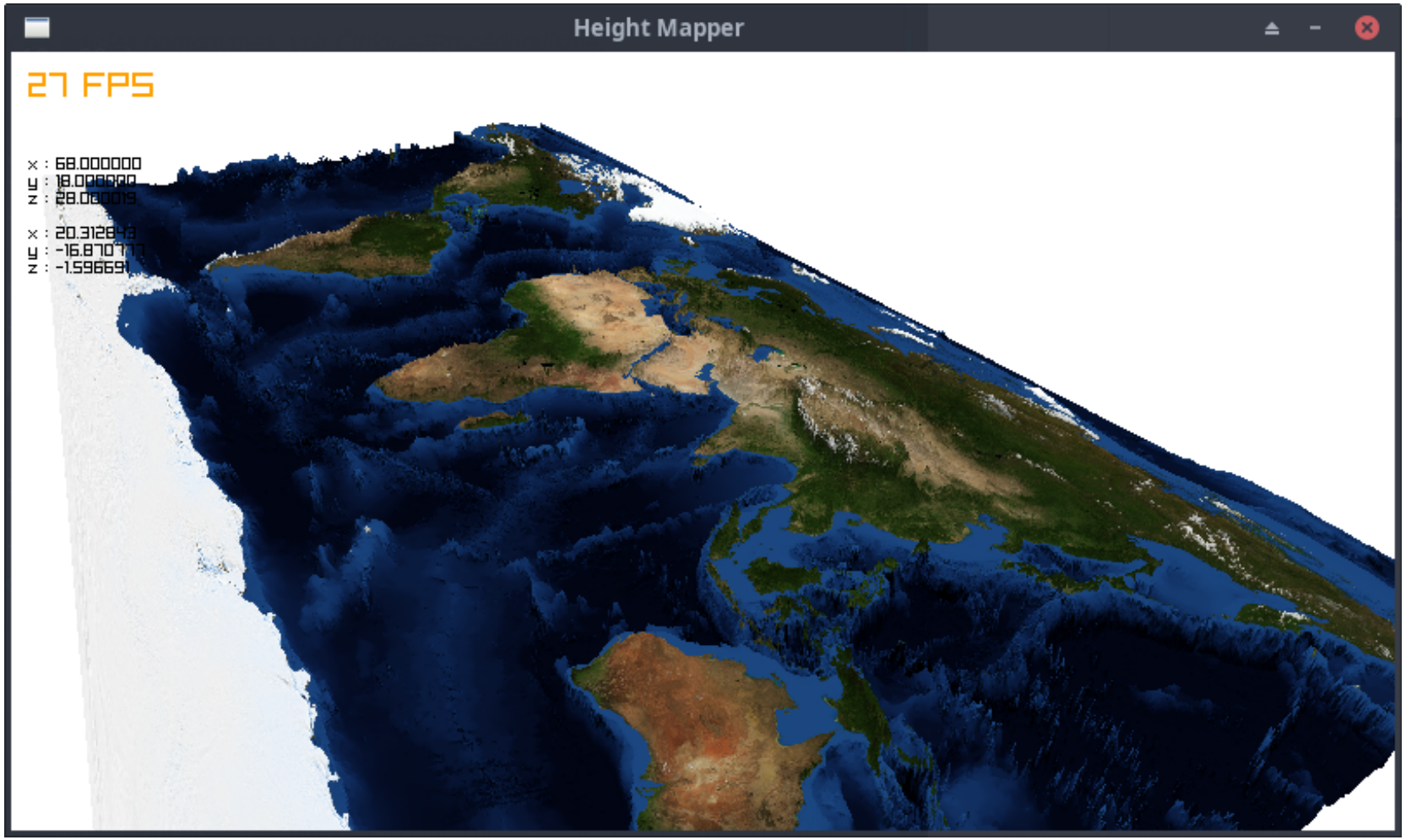


Figure: Rendu pour la figure de droite avec une texture (NASA dataset).

Problème

Problématique

Comment concilier réalisme et efficacité algorithmique dans la génération de cartes de hauteur, et quels compromis cela implique-t-il en termes d'applications pratiques ?

Algorithme : bruit de Perlin

Histoire : film *Tron* (1982), Ken Perlin cherche à générer des effets spéciaux au rendu plus naturel, moins "machinique" (cf. «*History* »).

- Technique de génération *pseudo-aléatoire* de textures.
- **Bruit de Perlin** (1981).

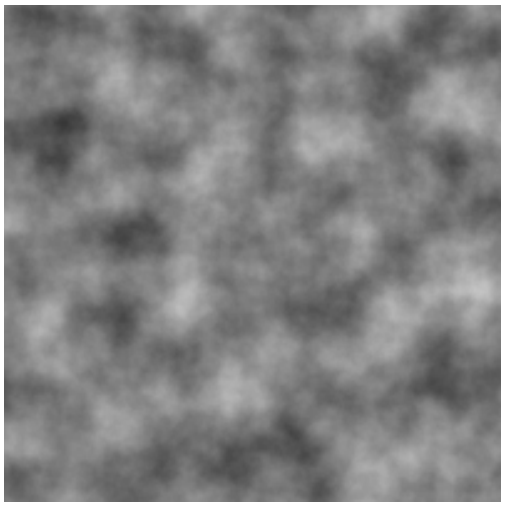
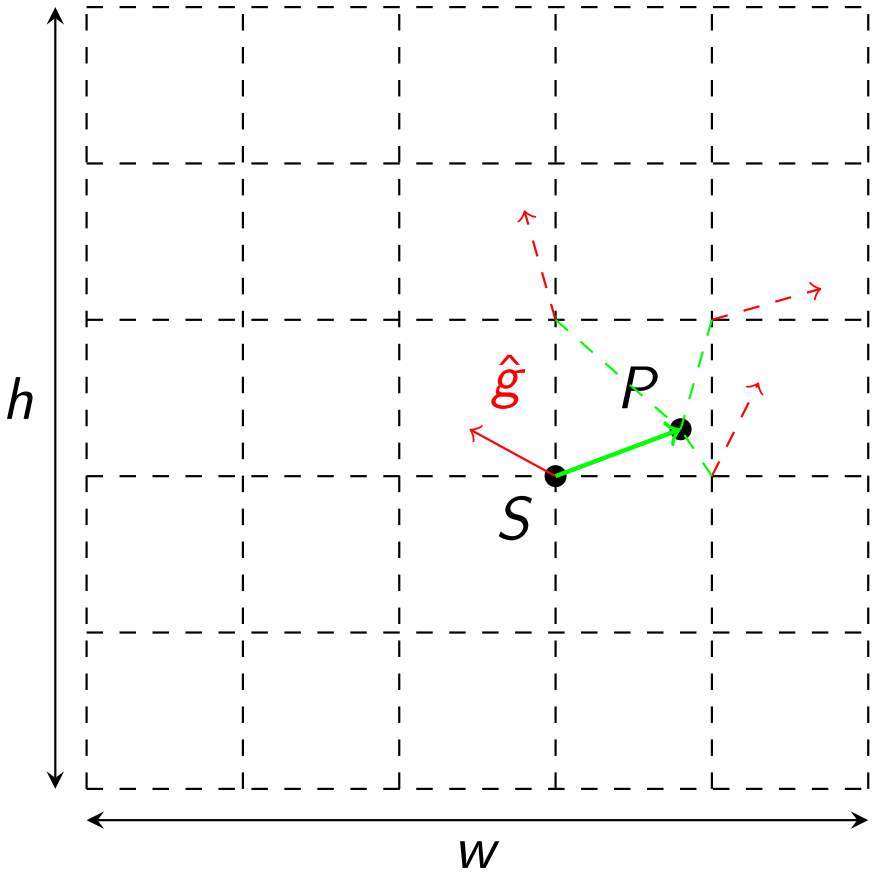


Figure: Rendu réussi (8 octaves).

Algorithme : bruit de Perlin

Grille de taille $w \times h$:



Étapes

Pour $P \in P_{w,h}$:

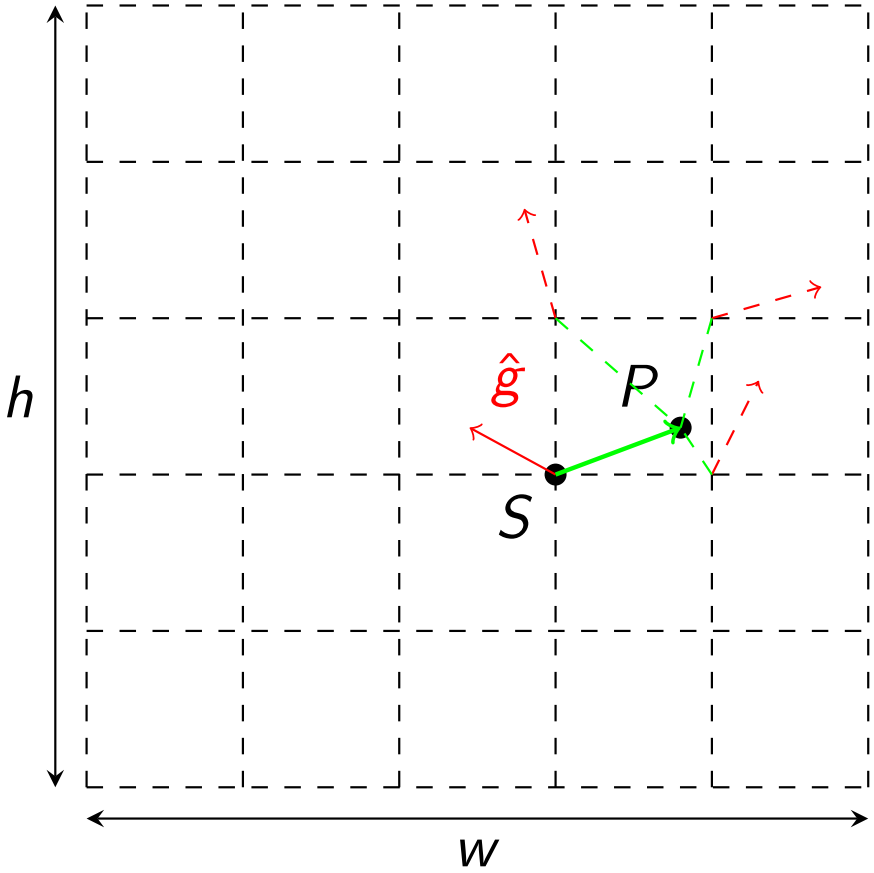
- Gradient aléatoire (unitaire) $\hat{g}$ associé à chaque sommet S de la grille.
- Calcul du produit scalaire $\hat{g} \cdot \overrightarrow{SP}$ et des 3 autres.
- Interpolation linéaire b entre ces 4 produits scalaires.

Résultat

b est le *bruit de Perlin* en P

Algorithme : bruit de Perlin

Grille de taille $w \times h$:

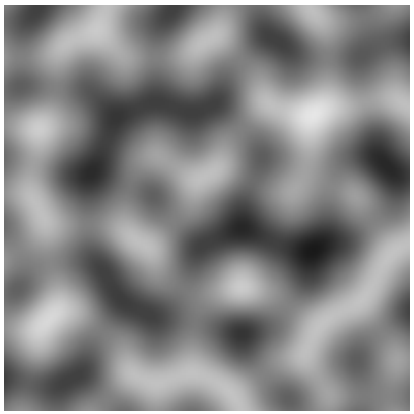


On répète le procédé pour chaque point de la carte.

Algorithme : bruit de Perlin

Bruit de Perlin fractal

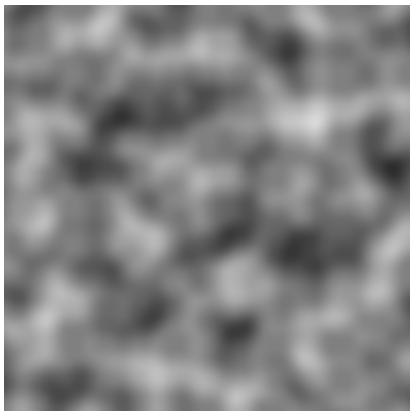
On fait varier la fréquence et l'amplitude des points sur la grille, o fois (octaves).



1 octave

$1 \times f = f$

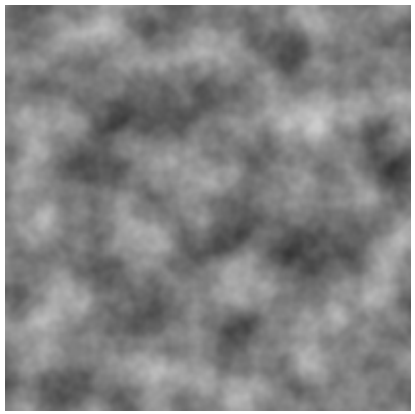
$\frac{A}{1} = A$



2 octaves

$2 \times f$

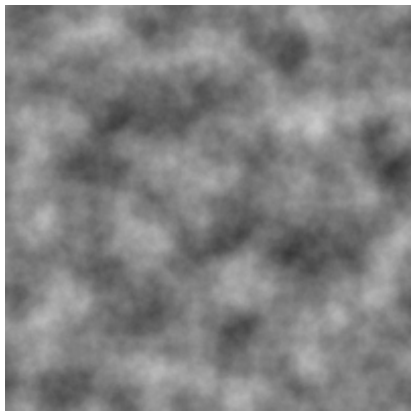
$\frac{A}{2}$



4 octaves

$4 \times f$

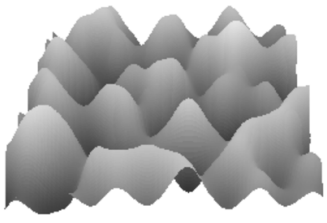
$\frac{A}{4}$



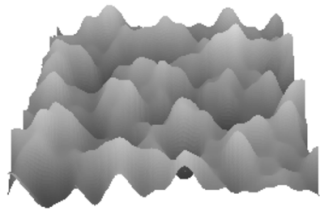
8 octaves

$8 \times f$

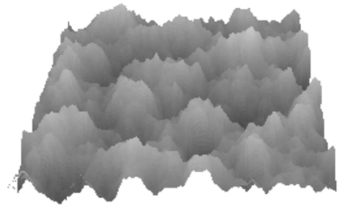
$\frac{A}{8}$



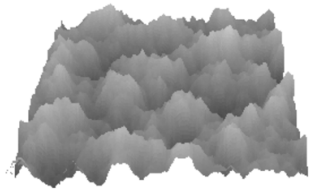
1 octave



2 octaves



4 octaves



8 octaves

Algorithme : filtre moyenneur

Qu'est-ce qu'un filtre moyenneur ?

Soit $H \in \mathcal{M}_{w,h}(L)$ et

$$K = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

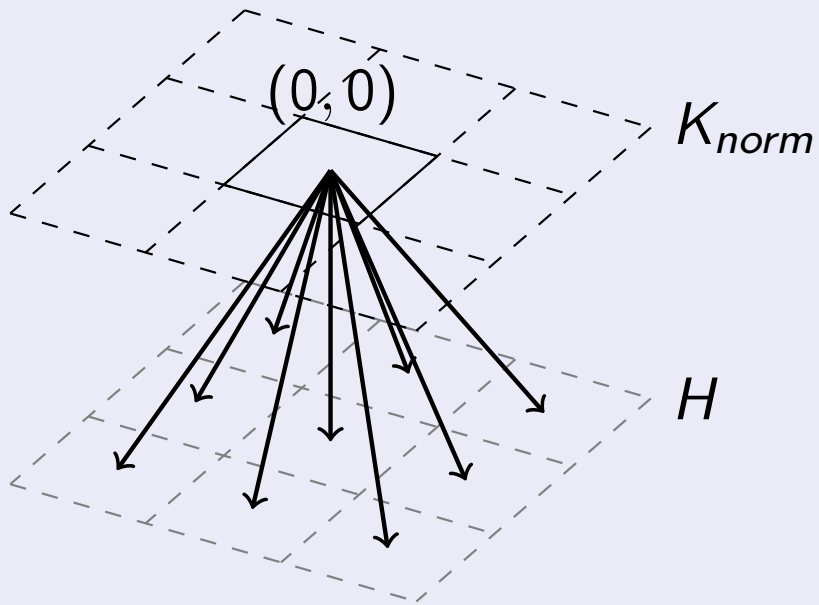
Alors

- $K_{norm} = \frac{1}{\sum_{i,j}[K]_{i,j}} K = \begin{pmatrix} \frac{1}{10} & \frac{1}{10} & \frac{1}{10} \\ \frac{1}{10} & \frac{1}{5} & \frac{1}{10} \\ \frac{1}{10} & \frac{1}{10} & \frac{1}{10} \end{pmatrix}$
- $[H_{moy}]_{i,j} = [H * K_{norm}]_{i,j} = h_{H*K_{norm}}(i,j)$

où : $\forall (i,j) \in P_{w,h} \cap (\mathbb{N}^*)^2$,

$$[H * K_{norm}]_{i,j} = \sum_{u=-1}^{1} \sum_{v=-1}^{1} [K_{norm}]_{u+2,v+2} \cdot [H]_{i+u,j+v}$$

Carte de hauteur moyenne

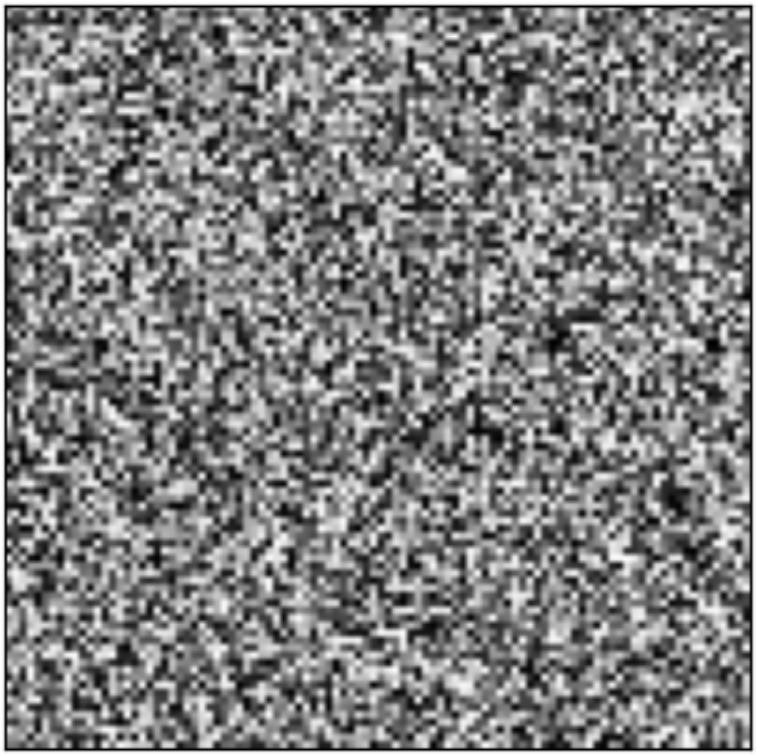


Algorithme : filtre moyenneur

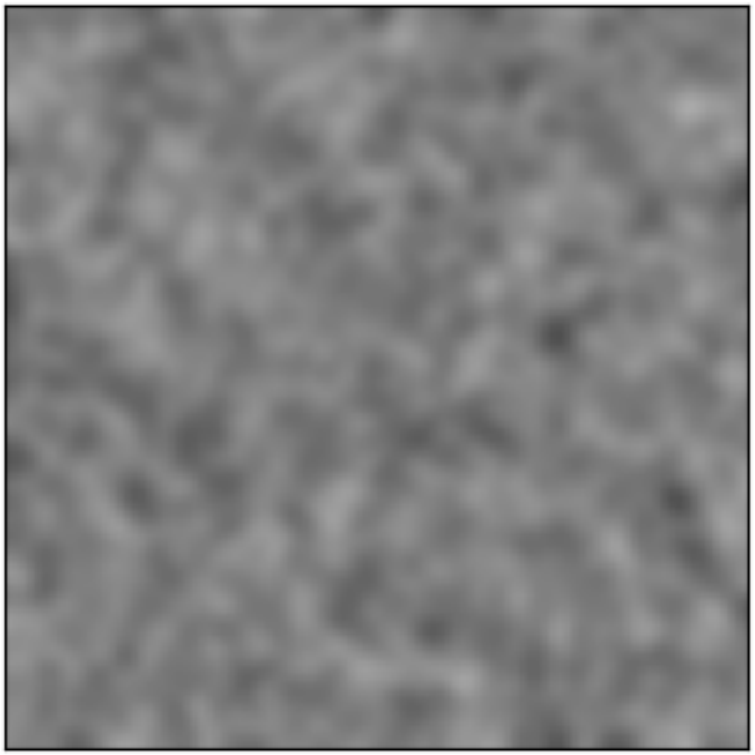
Utilisation d'un filtre moyenneur sur un bruit classique :

$\alpha = 0.5$, layers $= 10$

Carte initiale

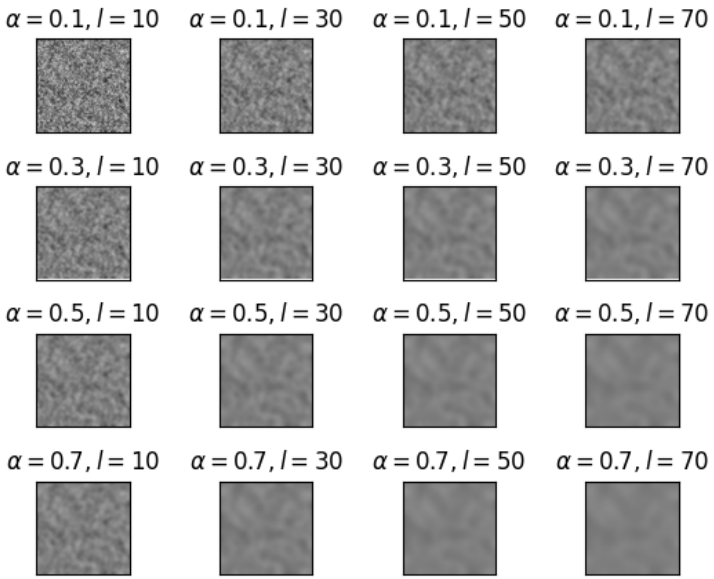


Carte après diffusion

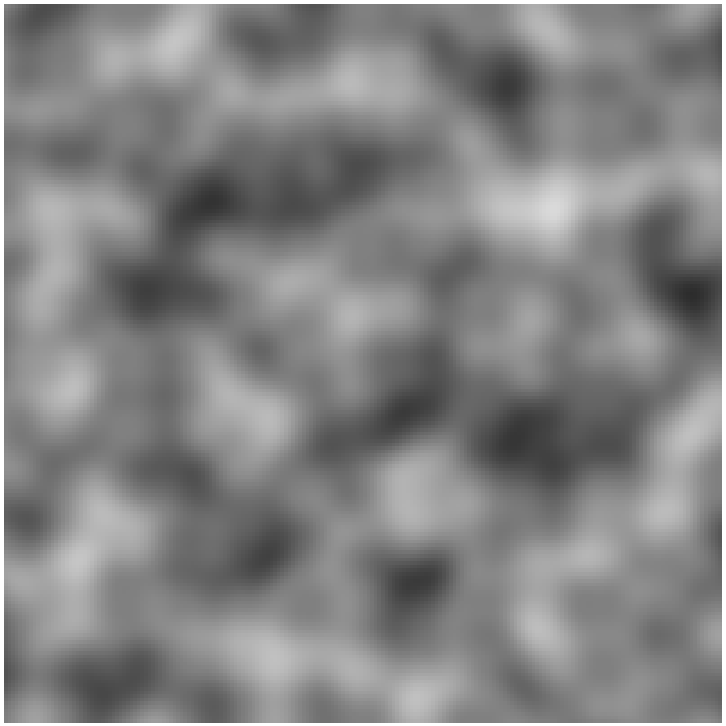


Algorithme : filtre moyenneur

En faisant varier les paramètres :

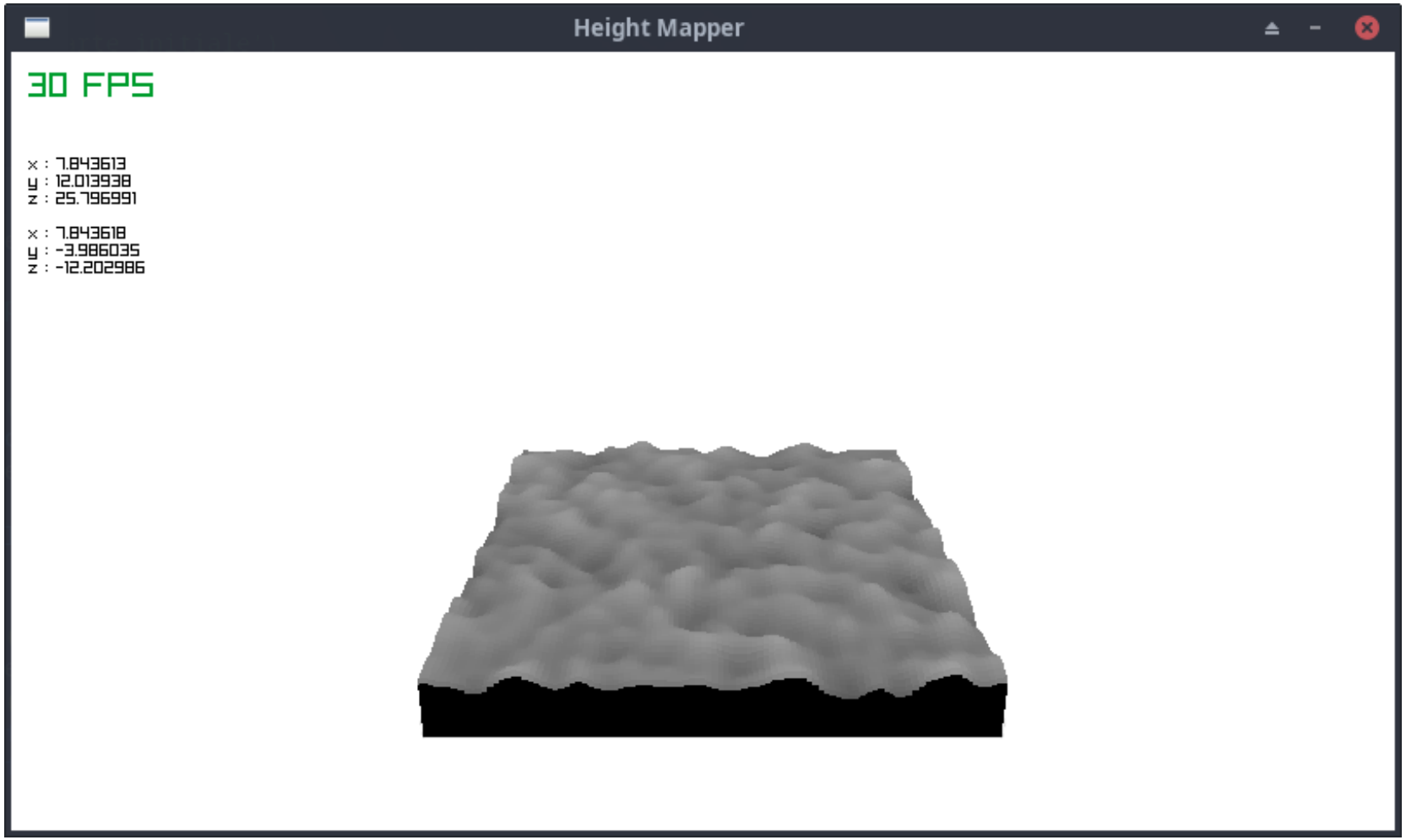


Objectif : un rendu similaire au bruit de Perlin à 2 octave.



Algorithme : filtre moyenneur

Rendu 3D :

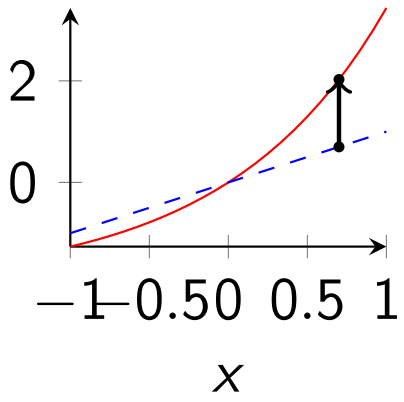


Algorithme : filtre moyenneur

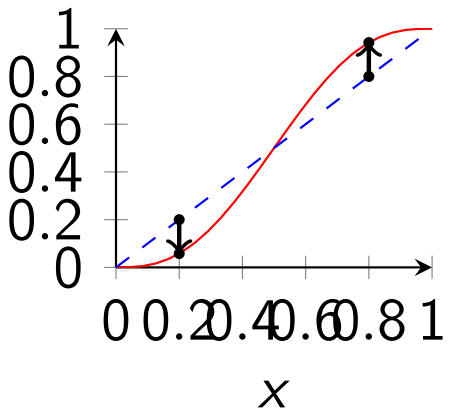
Amélioration 1 : ajustement du dénivelé

Fonctions définies sur $[-1, 1]$ permettant par un algorithme de modifier le dénivelé des montagnes. Exemples :

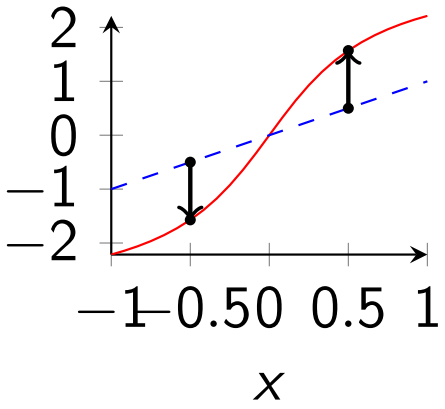
$2(\exp(x) - 1)$



$6x^5 - 15x^4 + 10x^3$



$2\arctan(2x)$

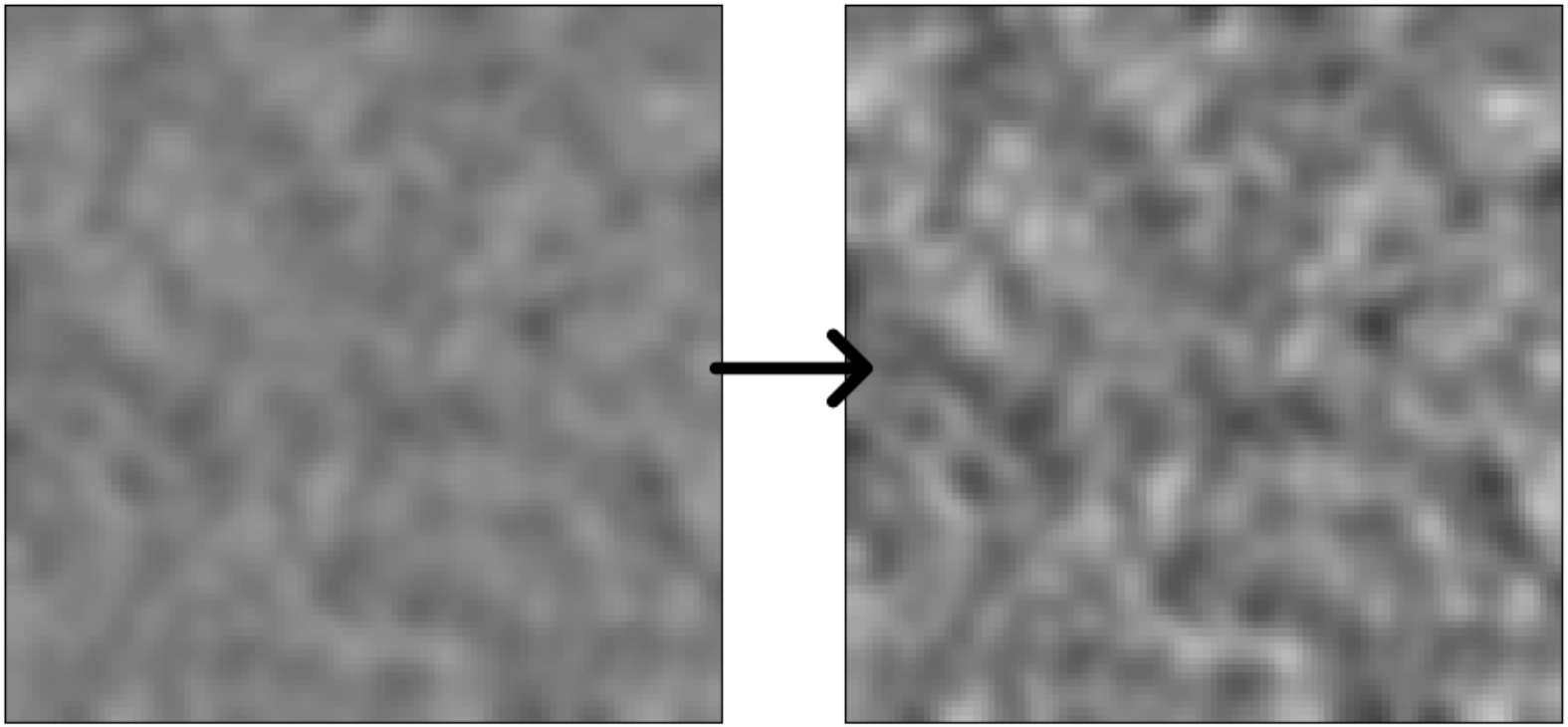


Permet de :

- Faire varier la taille des montagnes ;
- En modifier l'allure.

Algorithme : filtre moyenneur

Comparaison : pour $x \mapsto 2(\exp(x) - 1)$

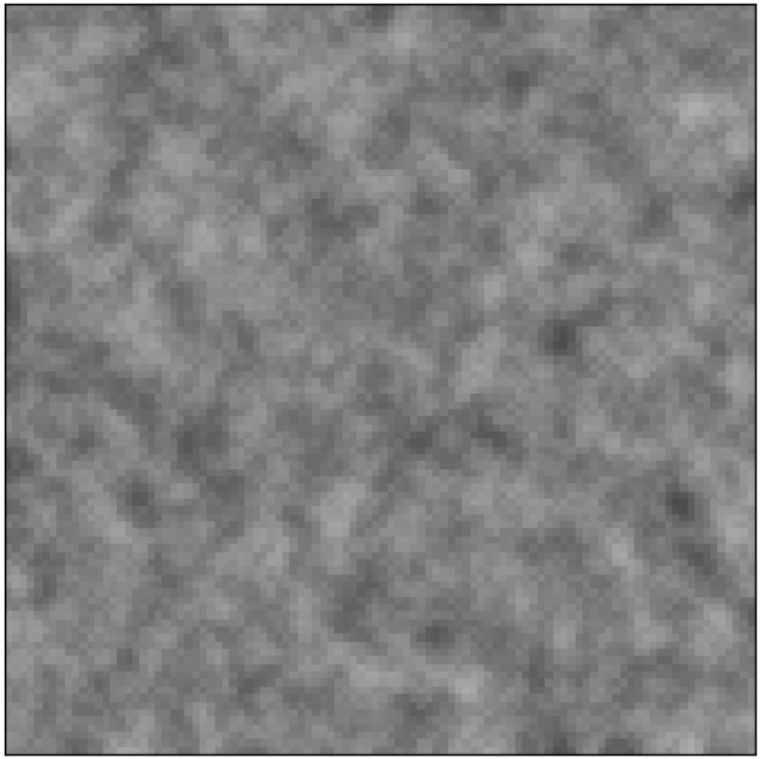


Algorithme : filtre moyenneur

Rendu 3D :

Algorithme : filtre moyenneur

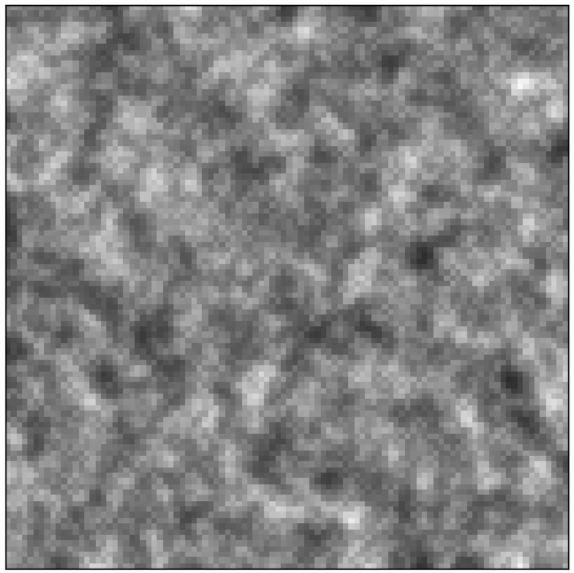
Amélioration 2 : fractalisation du bruit



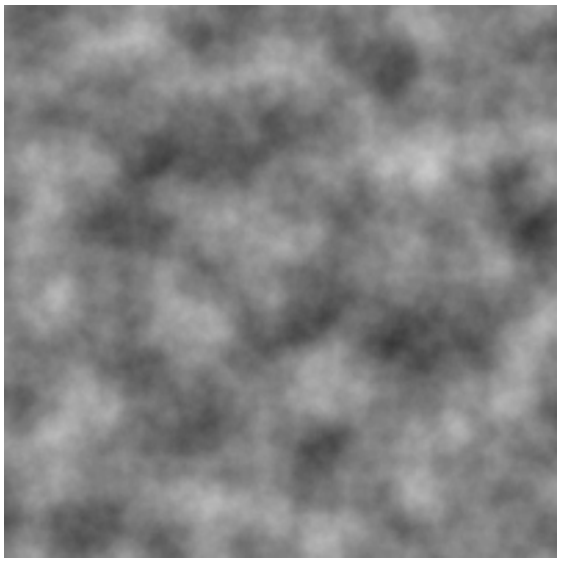
Algorithme : filtre moyenneur

Amélioration 2 : fractalisation du bruit

...donc en combinant avec l'ajustement du dénivelé



Filtre moyenneur
Amélioration 1
Amélioration 2

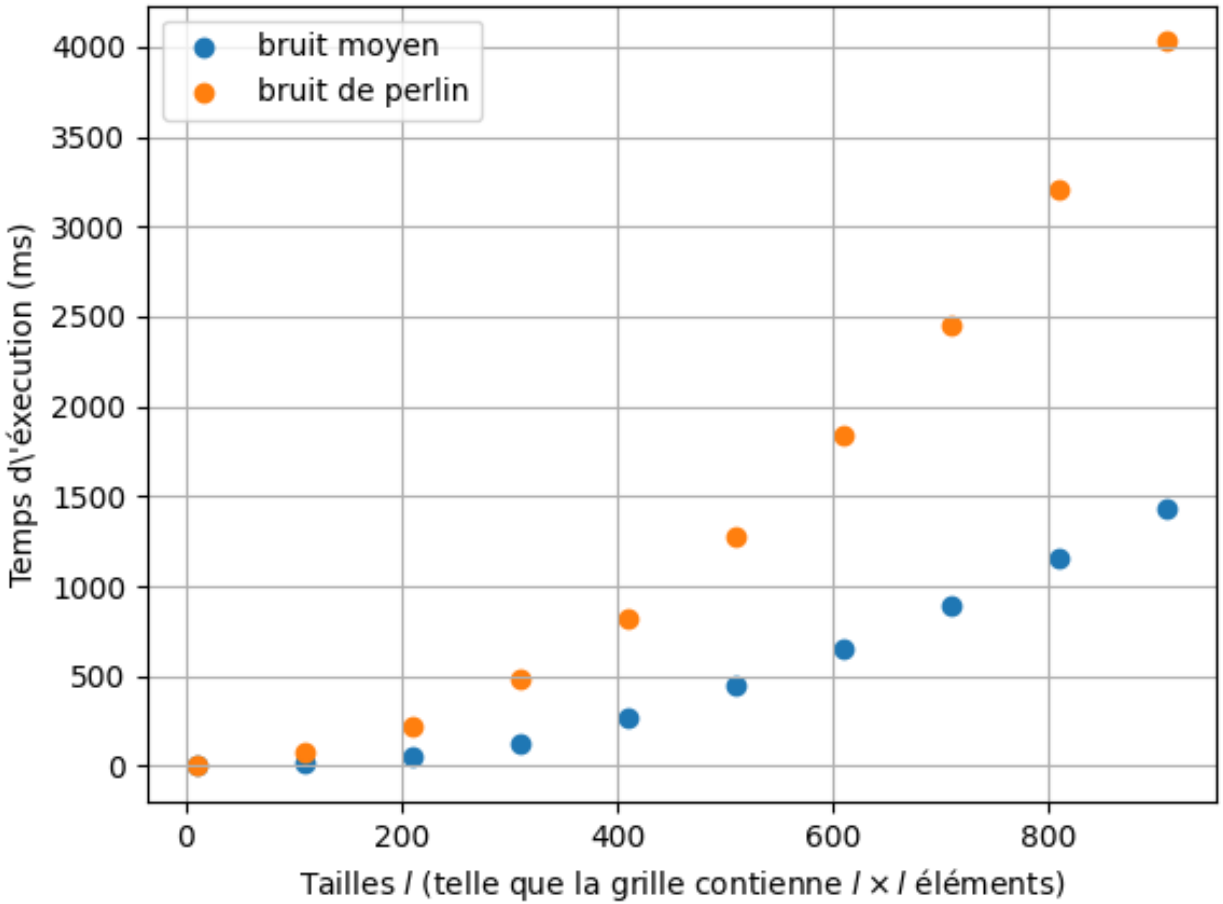


Rendu de Perlin à 8 octaves

$\Rightarrow$ Résultats proches mais un rendu peu satisfaisant.

Complexité temporelle et temps d'exécution

Bruit de Perlin : $O(\text{octaves} \times w \times h)$
Bruit moyenneur : $O(w \times h)$



Différences entre les deux algorithmes

Question

Pourquoi le bruit de Perlin reste largement plus utilisé ?

Avantages algo. #1

- Plus réaliste
- Plus grand contrôle (f, A, o)
- Déterministe (génération bout par bout)

Avantages algo. #2

- Plus facile à mettre en place
- Améliorations offrant un panel de configurations

Conclusion.

- Deux méthodes comparées : Perlin vs. filtrage moyenneur.
- Choix entre réalisme, efficacité algorithmique et usage pratique.
- Rendu final proche, mais performances et compromis différents.

Merci de m'avoir écouté !

Annexe 1.1. - bruit de Perlin

```python
import random
from PIL import Image
import numpy as np
from pathlib import Path

import matplotlib.pyplot as plt

Coord = tuple[int | float]
Vector = Coord
Gradient = Vector

Matrix = np.ndarray

def lerp(a: float, b: float, t: float) -> float:
    """ Complexité : O(1) """
    return a * (1 - t) + b * t

def fade(t: float) -> float:
    """ Complexité : O(1)
    A l'origine, utilisation de la fonction de lissage d'Hermite :
    3t^2-2t^3 mais celle-ci assure une 'meilleure' continuité. """
    return 6 * t ** 5 - 15 * t ** 4 + 10 * t ** 3 # Voir l'article de K. Perlin : "3. MODIFICATIONS"

def clamp(x: float, a: float = -1., b: float = 1.):
    """ Complexité : O(1) """
    return max(a, min(x, b))

def dot_product(u: Vector, v: Vector) -> float:
    """ Complexité : O(1) """
    return u[0] * v[0] + u[1] * v[1]
```

```python
class PerlinNoise(object):

    def __init__(self, width: int, height: int, octaves: int = 4, seed: int = -1) -> None:
        self.height = height
        self.width = width
        self.octaves = octaves

        if seed <= 0:
            seed = int(10000 * random.random())

        self.seed = seed

        sqrt2 = np.sqrt(2)
        self.gradients = [
            (1, 0), (-1, 0), (0, 1), (0, -1),
            (1 / sqrt2, 1 / sqrt2),
            (-1 / sqrt2, 1 / sqrt2),
            (1 / sqrt2, -1 / sqrt2),
            (-1 / sqrt2, -1 / sqrt2),
        ]

        # seed_x, seed_y = cantor_pairing_reciprocal(seed)
        # self.seed = (seed_x, seed_y)

        self.generated = None

    @classmethod
    def get_bounding_square(cls, x: float, y: float) -> list[Coord]:
        """ Renvoie les coins entiers du carré 1*1 délimitant (x, y)
            Complexité : O(1) """

        x_m, y_m = int(x), int(y)
        return [(x_m + i, y_m + j) for j in range(2) for i in range(2)]
```

```python
    def cell_lerp(self, x: float, y: float, corners: list[Coord]) -> float:
        """ Complexité : O(1) ; (cell linear interpolation) """
        dot_prods = self.get_dot_prods(x, y, corners)

        return lerp(
            lerp(
                dot_prods[corners[0]],
                dot_prods[corners[1]],
                fade(x - int(x))
            ),
            lerp(
                dot_prods[corners[2]],
                dot_prods[corners[3]],
                fade(x - int(x))
            ),
            fade(y - int(y))
        )

    def get_random_gradient(self, x: int, y: int) -> Gradient:
        """ Complexité : ~O(1) ; dépendent de x et y car l'aléatoire doit dépendre de la position (caractère déterministe du bruit de Perlin) """

        h = hash((x, y, self.seed))
        index = h % len(self.gradients)
        return self.gradients[index]

    def get_dot_prods(self, x: float, y: float, corners: list[Coord]) -> dict[tuple, float]:
        """ Complexité : O(1) """
        dot_prods = {}

        for (i, j) in corners:
            v = (x - i, y - j)
            g = self.get_random_gradient(i, j)
            dot_prods[(i, j)] = dot_product(v, g)

        return dot_prods
```

```python
    def __noise(self, x: float, y: float) -> float: # Noise function
        """ Complexité : O(1) """
        corners = self.get_bounding_square(x, y)
        interpolation = self.cell_lerp(x, y, corners)

        return interpolation

    def __iterated_noise(self, x: float, y: float, amplitude: float = 1.0,
                         frequency: int = 1, persistence: float = 0.5) -> float:
        """ Complexité : O(octaves) """
        res = .0
        tot_amplitude = .0

        for _ in range(self.octaves):
            res += self.__noise(x * frequency, y * frequency) * amplitude
            tot_amplitude += amplitude
            amplitude *= persistence
            frequency *= 2

        return res / tot_amplitude

    def __noise_to_grayscale(self, noise: float) -> int:
        """ Transforms a noise value (from -1 to 1) to a grayscale value (from 0 to 255) """
        return int(clamp(noise) * 127) + 128
```

```python
    def pixel_matrix(self, initial_scale: float = .03,
                     print_trace: bool = False) -> Matrix:
        """ Complexité : O(octaves * width * height) """

        matrix = []
        for j in range(self.height):
            line = []
            for i in range(self.width):
                line.append(
                    self.__noise_to_grayscale(
                        self.__iterated_noise(i * initial_scale, j * initial_scale)
                    )
                )
            matrix.append(line)

        return np.array(matrix)

    def generate(self, print_trace: bool = True) -> np.ndarray:
        print(f'Heightmap (octaves = {self.octaves}, size = ({self.width}, \
              {self.height})) is being generated.')

        m = self.pixel_matrix(print_trace = print_trace)
        self.generated = m
        return m
```

```python
    def save(self, out: str | Path) -> None:
        if self.generated is None:
            self.generate()

        img = Image.fromarray(np.uint8(self.generated), 'L')
        img.save(out)

    def show(self) -> None:
        if self.generated is None:
            self.generate()

        plt.imshow(self.generated, cmap = "gray", vmin = 0, vmax = 255)
        plt.show()

if __name__ == '__main__':

    for i in range(3 + 1):
        o = 2 ** i
        pn = PerlinNoise(200, 200, o, 12812)
        name = f'perlin_{o}octaves'
        pn.save(out = f'./out/perlin_noise/{name}.png')
        pn.save(out = f'../ocaml/resources/{name}.png')
```

Annexe 1.2. - bruit par filtre moyenneur

```python
import numpy as np
from numpy.random import RandomState, SeedSequence, MT19937
from numpy._typing import _ufunc

import matplotlib.pyplot as plt
from matplotlib.axis import Axis

from scipy import ndimage

heightmap = np.ndarray

class SmoothedNoise(object): # "bruit lissé"

    def __init__(self, size: tuple[int], seed: int = -1) -> None:
        self.size = size

        # Nouveau générateur de nombres aléatoires selon seed (voir doc. de la f° np.random.seed) :

        sseq = SeedSequence(seed)
        if sseq == -1:
            sseq = SeedSequence()

        self.rg = RandomState(MT19937(sseq)) # "rg" : Random Generator

        self.map = self.__generate_random_map()

    def __generate_random_map(self) -> np.ndarray:
        return self.rg.randint(0, 255, size = self.size)
```

```
    def generate_filtered_map(self, ker: np.ndarray, alpha: float = 0.5,
                              layers: int = 1, shrink: _ufunc = None) -> np.ndarray:

        h = self.map

        # Normalisation du noyau
        ker = ker / ker.sum()

        for _ in range(layers):
            smoothed = ndimage.convolve(h, ker)

            # Interpolation entre nouvelle et ancienne carte
            h = (1 - alpha) * h + alpha * smoothed

        if shrink:
            h = self.shrink_render(h, func = shrink)

        return h

    def generate_fractal_filtered_map(self, ker: np.ndarray, alpha: float = .5,
                                      octaves: int = 4, shrink: _ufunc = None) -> np.ndarray:

        layers = 1
        h = self.generate_filtered_map(ker, alpha, layers, shrink)
        for i in range(1, octaves + 1):
            h = alpha * h + (1 - alpha) * self.generate_filtered_map(ker, alpha, 2 ** i, shrink)

        return h

    @classmethod
    def __remove_axes_text(cls, ax: Axis) -> None:
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```

```python
def __show_map(self, h: np.ndarray, title: str, save: bool = False, show: bool = True, show_initial_map: bool = False) -> None:

    # Ferme la figure courante, si elle existe.
    plt.close()

    if show_initial_map:
        fig, axes = plt.subplots(1, 2, tight_layout=True)
        a1, a2 = axes

        #a1.set_title('Carte initiale')
        a1.imshow(self.map, cmap = 'gray')
        self.__remove_axes_text(a1)

        #a2.set_title('Carte après diffusion')
        a2.imshow(h, cmap = 'gray', vmin = 0, vmax = 255)
        self.__remove_axes_text(a2)
    else:
        fig, axes = plt.subplots(1, 1, tight_layout=True)
        a = axes

        #a.set_title('Carte après diffusion')
        a.imshow(h, cmap = 'gray', vmin = 0, vmax = 255)
        self.__remove_axes_text(a)

    fig.suptitle(title)

    if save:
        name = input('Quel nom donner a ce rendu ? \n > ')
        fig.set_frameon(False)
        fig.suptitle('')
        fig.savefig(f'./out/naive_noise/{name}.png')
        fig.savefig(f'../ocaml/resources/{name}.png')

    if show:
        plt.show()
```

```python
    def show_filtered_map(self, ker: np.ndarray, alpha: float = .5, layers: int = 1, shrink: _ufunc = None, save: bool = False, show: bool = True) -> None:

        h = self.generate_filtered_map(ker, alpha, layers, shrink)
        title = fr'Filtre : $\alpha={alpha}$, layers$={layers}$'

        self.__show_map(h, title, save, show)

    def show_range_of_results(self, ker: np.ndarray, save: bool = False, shrink: _ufunc = None) -> None:
        l_alpha = [.1, .2, .3, .4, .5, .6, .7, .8, .9]
        l_layers = [10, 20, 30, 40, 50, 60, 70, 80, 90]

        n = len(l_alpha)
        m = len(l_layers)

        plt.close()

        fig, axes = plt.subplots(n, m, tight_layout = True)

        fig.set_facecolor('black')
        fig.suptitle(r'Grille de résultats faisant varier $\alpha$ et le nombre de couches (layers)', color = 'white')

        for i in range(n):
            alpha = l_alpha[i]
            for j in range(m):
                layers = l_layers[j]

                h = self.generate_filtered_map(ker, alpha, layers, shrink)

                a = axes[i, j]
                # a.set_title(fr'$\alpha={alpha}$, layers$={layers}$')
                a.imshow(h, cmap = 'gray', vmin = 0, vmax = 255)
                self.__remove_axes_text(a)


        if save:
            name = input('Quel nom donner a ce rendu ? \n > ')
            fig.savefig(f'./out/naive_noise/comparisons/{name}.png')

        plt.show()
```

```python
    def show_fractal_filtered_map(self, ker: np.ndarray, alpha: float = .5, octaves: int = 4,
                                  shrink: _ufunc = None, save: bool = False,
                                  show: bool = True) -> None:

        h = self.generate_fractal_filtered_map(ker, alpha, octaves, shrink)
        title = fr'Filtre + Fractal : $\alpha={alpha}$, octaves$={octaves}$'

        self.__show_map(h, title, save, show)

     def shrink_render(self, m: np.ndarray, func: _ufunc = np.tan) -> np.ndarray:
        """ Le rendu seul ayant un 'dénivelé' trop important, il est nécessaire de réduire
            l'écart-type de hauteur : cette fonction s'en charge. """

        m_norm = m / 128
        m_shrink = func(m_norm - 1) + 1

        return m_shrink * 128

    def sharpen_render(self, h: np.ndarray) -> np.ndarray:
        """ Le rendu seul étant trop 'lisse', il est nécessaire d'aiguiser les sommets
            (sections sombres) pour le rendre plus naturel : cette fonction s'en charge. """

        ...
```

```python
if __name__ == '__main__':
    ker = np.array(
        [[1, 1, 1],
         [1, 2, 1],
         [1, 1, 1]]
    )

    sn = SmoothedNoise((1000, 1000), 1)
    sn.show_filtered_map(
        ker = ker,
        layers = 150,
        alpha = .8,
        save = False,
        show = True,
        shrink = lambda m : 5 * (np.exp(m) - 1)    #lambda x : (np.exp(2 * x - 1.3) - 0.5)
                                                   #lambda m : 0.5 * np.tan(m)
    )

    sn.show_range_of_results(ker, save = False)

    sn.show_fractal_filtered_map(
        ker,
        octaves = 4,
        save = False,
        show = True,
        shrink = lambda m : 2 * (np.exp(m) - 1)

```

Annexe 1.3. - calcul de temps d'exécution

```python
import matplotlib.pyplot as plt
import numpy as np

from ext import timeit, ns_to_ms

from naive_noise import SmoothedNoise
from perlin_noise import PerlinNoise

from typing import Callable

SN_KER = ker = np.array(
    [[1, 1, 1],
    [1, 2, 1],
    [1, 1, 1]]
)

SN_ALPHA = 0.5

def compute_simple_generation(size: tuple[int], sn_ker: np.ndarray, sn_alpha: float = 0.5, avgnoise: bool = True, perlinnoise: bool = True) -> tuple[float]:
    """ 'size' : (width, height,) --> taille de la carte générée.
        'sn_ker' : noyau choisi pour la génération de la carte par filtre moyen.
        'sn_alpha' : proportion d'interpolation choisie pour la carte générée par filtre moyen.

        - Génération d'une carte par chacun des algorithmes dans leur version la plus légère (aucun octave, aucune amélioration supplémentaire pour 'SmoothedNoise', etc.)

        - Pour chacun, la graine est générée aléatoirement (pas de graine fixe).

        Renvoie le nombre de millisecondes de calculs pour chacun des deux algorithmes, sous forme de tuple.
    """

    ... # suite sur la page suivante...
```

```
    ... # suite ...

    w, h = size

    ## Instances

    if avgnoise:
        sn = SmoothedNoise(
            size = size,
        )

    if perlinnoise:
        pn = PerlinNoise(
            width = w,
            height = h,
            octaves = 1,
        )

    ## Calculs des temps

    t_sn, t_pn = None, None

    if avgnoise:
        t_sn = ns_to_ms(timeit(
            f = sn.generate_filtered_map,
            ker = sn_ker,
            alpha = sn_alpha,
            layers = 150
        ))

    if perlinnoise:
        t_pn = ns_to_ms(timeit(
            f = pn.generate,
        ))

    return t_sn, t_pn
```

```python
def graph_complexity(sizes_nb: int, step: Callable) -> None:

    sizes = [step(size) for size in range(sizes_nb)]

    sn_exec_times = []
    pn_exec_times = []

    for size in sizes:
        t_sn, t_pn = compute_simple_generation(
            size = (size, size),
            sn_ker = SN_KER,
            sn_alpha = SN_ALPHA
        )

        sn_exec_times.append(t_sn)
        pn_exec_times.append(t_pn)

    plt.scatter(sizes, sn_exec_times, label = 'bruit moyen')
    plt.scatter(sizes, pn_exec_times, label = 'bruit de perlin')

    plt.legend()

    plt.xlabel(r'Tailles $l$ (telle que la grille contienne $l\times l$ éléments)')

    plt.ylabel(r'Temps d\'éxecution (ms)')

    plt.grid()
    plt.show()

if __name__ == '__main__':
    graph_complexity(10, lambda t : 10 + 100 * t)
```

Annexe 2.1. - Rendu 3D (Raylib)

```
type heightmap = {
  scale: float;
  width: float;
  height: float;
  texture: Raylib.Texture2D.t;
  mesh: Raylib.Mesh.t;
  model: Raylib.Model.t
}[@@warning "-69"]

type state = {
  camera : Raylib.Camera3D.t;
  hm : heightmap;
  map_pos : Raylib.Vector3.t
}

type coordinates = {
  x : float;
  y : float;
  z : float
}

let load_heightmap (img: Raylib.Image.t) (tex: Raylib.Image.t option) =
  let open Raylib in
    let scale = 1. /. 10. in
    let width = float_of_int (Image.width img) *. scale
    and height = float_of_int (Image.height img) *. scale in

    let texture = match tex with
      | Some t -> load_texture_from_image t
      | None -> load_texture_from_image img
    in

    let mesh = gen_mesh_heightmap
      img
      (Vector3.create width 10. height)
    in
    let model = load_model_from_mesh mesh in
    {scale; width; height; texture; mesh; model}

```

```ocaml
let draw_coordinates (coords: coordinates) x y z =
  let open Raylib in
    draw_text (Printf.sprintf "x : %f" (coords.x)) x y z Color.black;
    draw_text (Printf.sprintf "y : %f" (coords.y)) x (y + 10) z Color.black;
    draw_text (Printf.sprintf "z : %f" (coords.z)) x (y + 20) z Color.black

let init_transform_from_path (screen_size: int * int) (hm_img_path: string) (texture_path: string option) =
  let open Raylib in
    init_window
      (fst screen_size)
      (snd screen_size)
      "Height Mapper";
    let camera =
      Camera.create
        (* Position *)
        (Vector3.create 8. 13. 31.)
        (* Target *)
        (Vector3.create 8. (-3.) (-7.))
        (* Rotation over X axis *)
        (Vector3.create 0. 1. 0.)
        (* FOV aperture in Y, degrees *)
        45.
        (* Projection (perspective/orthographic) *)
        CameraProjection.Perspective
    in
    (* let hm_texture = load_image (base ^ texture) in *)
    let hm_img = load_image hm_img_path in
    let texture_img = match texture_path with
      | Some path -> if path = "" then None else Some (load_image path)
      | None -> None
    in
    let hm = load_heightmap hm_img texture_img in
    let map_pos = Vector3.create 0. 0. 0. in
    set_material_texture
      (CArray.get (Model.materials hm.model) 0 |> addr)
      MaterialMapIndex.Albedo hm.texture;
    unload_image hm_img;
    disable_cursor ();
    set_target_fps 30;
    { camera; hm; map_pos }
```

```
let init_transform_from_array (screen_size: int * int) (hm_img_array: int array array) (texture_path: string option) =
  let open Raylib in
    init_window
      (fst screen_size)
      (snd screen_size)
      "Height Mapper";
    let camera =
      Camera.create
        (* Position *)
        (Vector3.create 8. 13. 31.)
        (* Target *)
        (Vector3.create 8. (-3.) (-7.))
        (* Rotation over X axis *)
        (Vector3.create 0. 1. 0.)
        (* FOV aperture in Y, degrees *)
        45.
        (* Projection (perspective/orthographic) *)
        CameraProjection.Perspective
    in
    (* let hm_texture = load_image (base ^ texture) in *)
    let hm_img = Heightmapper.create_image_from_grayscale hm_img_array in
    let texture_img = match texture_path with
      | Some path -> if path = "" then None else Some (load_image path)
      | None -> None
    in
    let hm = load_heightmap hm_img texture_img in
    let map_pos = Vector3.create 0. 0. 0. in
    set_material_texture
      (CArray.get (Model.materials hm.model) 0 |> addr)
      MaterialMapIndex.Albedo hm.texture;
    unload_image hm_img ;
    (* disable_cursor (); *)
    disable_cursor ();
    set_target_fps 30;
    { camera; hm; map_pos }
```

```
let close_win state =
  let open Raylib in
    unload_texture state.hm.texture;
    unload_model state.hm.model;
    Raylib.close_window ()

let rec loop state =
  (* Detect window close button or ESC key *)
  if Raylib.window_should_close () then
    state
  else
    let open Raylib in
      update_camera (addr state.camera) CameraMode.Free;
      begin_drawing();
        clear_background Color.white;
        begin_mode_3d state.camera;
          draw_model state.hm.model state.map_pos 0.25 Color.white;
        end_mode_3d ();
        draw_fps 10 10;
        let cam_pos = Camera.position state.camera in
        let cam_coords = {
          x = (Vector3.x cam_pos);
          y = (Vector3.y cam_pos);
          z = (Vector3.z cam_pos)
        } in
        let cam_target = Camera.target state.camera in
        let cam_target_coords = {
          x = (Vector3.x cam_target);
          y = (Vector3.y cam_target);
          z = (Vector3.z cam_target)
        } in
        draw_coordinates cam_coords 10 60 0;
        draw_coordinates cam_target_coords 10 100 0;
      end_drawing ();

      loop state
```

```
let run_from_path (screen_size: int * int) (hm_img_path: string) (texture_path: string option) =
  init_transform_from_path screen_size hm_img_path texture_path |> loop |> close_win
;;

let run_from_noise (screen_size: int * int) (seed: int) (texture_path: string option) =
  let x, y = screen_size in
  let hm_img_array = Heightmapper.noise seed x y 0.06 in
  init_transform_from_array screen_size hm_img_array texture_path |> loop |> close_win
;;
```

```
let print_matrix matrix =
  Array.iter (fun row ->
    Array.iter (fun elem ->
      Printf.printf "%d " elem (* Adjust formatting for element type, e.g., `%d` for integers *)
    ) row;
    Printf.printf "\n"
  ) matrix
;;

let launch_3d_render () =
  let sw = 800 in
  let sh = 450 in
  let base = "/run/media/gabrield/USB-128/workspace/dev/tipe-heightmaps/ocaml/resources/" in
  let heightmap = "with_shrink_exp_fractal.png" in
  (* let texture = "" in *)

  let open Mapper in
    Transform3d.run_from_path (sw, sh) (base ^ heightmap) (None)
    (* Transform3d.run_from_noise (sw, sh) 2819311 (Some texture) *)

  ;;

let () = launch_3d_render () ;;
```

Bibliographie (1)

- Perlin K. :
  - Algorithme : https://mrl.cs.nyu.edu/ perlin/noise/
  - Papier : https://mrl.cs.nyu.edu/ perlin/paper445.pdf
- Green S. : https://developer.nvidia.com/gpugems/gpugems2/part-iii-high-quality-rendering/chapter-26-implementing-improved-perlin-noise
- Peytavie A. : Génération procédurale de monde. https://theses.hal.science/tel-00841373/document (1.1.1.1 / 1.1.2.1)
- Nasa Visible Earth : https://visibleearth.nasa.gov/collection/1484/blue-marble?page=2

Bibliographie (2)

Librairies :

- OCaml :
  - Graphics : https://ocaml.org/p/graphics/latest
  - Raylib : https://ocaml.org/p/raylib/latest
- Python (avant amélioration de calcul) :
  - Numpy
  - Pillow (PIL)
  - librairies natives...