Introduction au bruit de Perlin

Jérémy Cochoy

10 février 2011

Résumé

Ken Perlin, dans le cadre de la réalisation du film *Tron* (1982) comportant des scènes en image de synthèse, se retrouva confronté à la limitation de la mémoire des machines de l'époque, ne permettant pas de stocker d'importantes et nombreuses textures.

Il chercha alors à générer ces textures par le calcul, que l'on nome *textures paramétriques*. Ceci l'amena à concevoir son algorithme homonyme qui sera le fil directeur de ce document.

Nous découvrirons en trois étapes son fonctionnement tout en construisant une implémentation.

Ce document s'adresse aux étudiants ayant assimilé le programme de terminale et expérimenté un premier contact avec la programmation. Quelques informations supplémentaires figurent toutefois pour les plus expérimentés ainsi que deux chapitres additionnels à la fin de ce document, où sont vaguement utilisées quelques notions de première année de licence. Elles ne sont toute fois pas nécessaires à la compréhension générale de ce document et des algorithmes présentés.

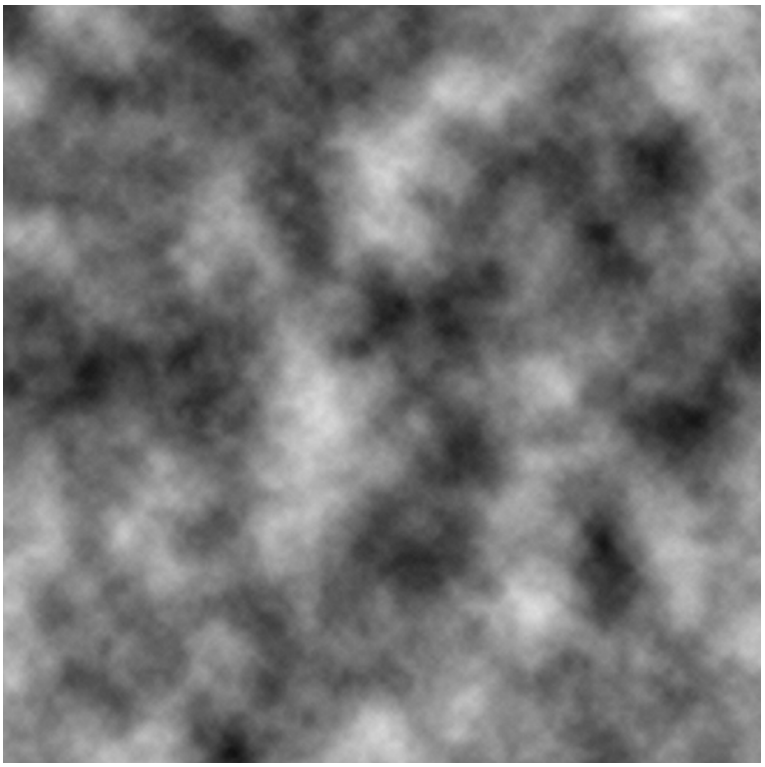
Table des matières

1 Introduction

L'algorithme de Perlin est à l'origine de nombreuses textures paramétriques, comme le marbre, le bois, l'effet de nuages, de brume, et de nombreux autres. Mais il est aussi utilisé pour la génération de reliefs et de terrains, la génération d'irrégularités dans la surface de solides (bump mapping) et de nombreuses autres applications.

Nous nous contenterons de discuter de son application à la génération de textures paramétriques, comme le présente la figure 1 basée sur un bruit bidimensionnel.

FIGURE 1 – Bruit de Perlin 2D



Dans ces premières parties, nous étudierons une version simplifiée de cet algorithme. Il se décompose en trois grandes parties :

— Une fonction de bruit qui permet d'associer des valeurs aléatoires à chaque point de l'espace $\mathbb{N}^{dim}$
— Une fonction de bruit lissé par interpolation
— Une combinaison linéaire de fonctions de bruit lissé

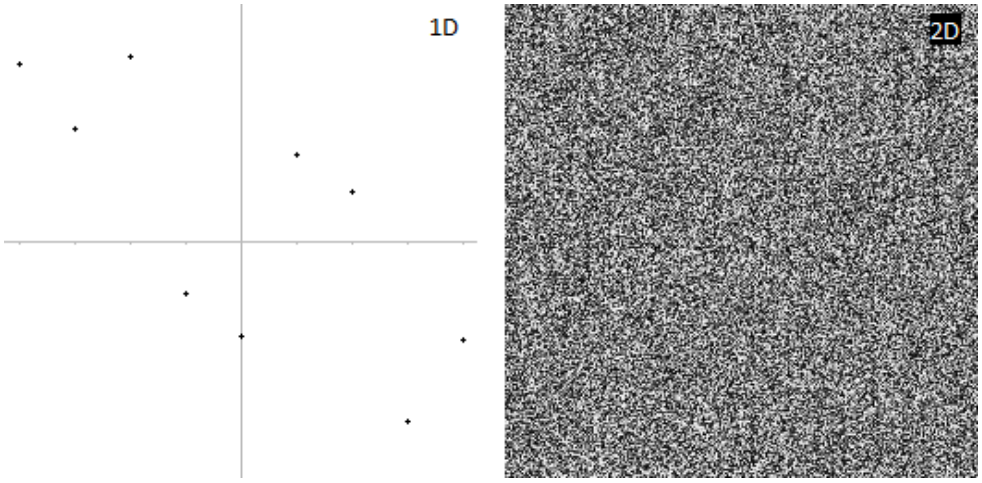
2 Génération d'un bruit

2.1 Fonctions pseudo-aléatoires

La première étape de l'algorithme consiste à former une fonction de bruit, c'est-à-dire une fonction $f : \mathbb{N} \to \mathbb{R}$ qui, à une valeur donnée, associe une valeur qui *semble aléatoire*. Il est important de remarquer que ce bruit pourra donc être reproduit, pour une même valeur. C'est très important, car si l'on souhaite par exemple l'utiliser pour générer une image, celle ci ne doit pas systématiquement changer, à chaque génération – à moins que ce paramètre ne soit voulu et contrôlé, ce dont nous reparlerons dans la partie suivante.

Vous avez un aperçu de bruit unidimensionnel et bidimensionnel avec la figure 2.

FIGURE 2 – Bruit 1D et 2D



Il est difficilement possible de produire de réelles valeurs aléatoires, et difficile pour un observateur de discerner un motif, une fois une certaine complexité ateinte. On se contentera donc de fonctions déterministes, périodiques, mais au rendu "désordonné".

Il nous faudra donc obtenir un jeu de valeurs aléatoires, dont le cycle – c'est-à-dire l'intervale nécessaire pour obtenir à nouveau le même jeu de valeurs – est assez important pour qu'un observateur ne puisse s'en rendre compte.

Vous pouvez utiliser une fonction de génération aléatoire fournie par le language que vous utilisez, comme la fonction *rand* du language C, et générer un tableau de valeurs, ou encore utiliser votre propre fonction pseudo-aléatoire.

Voici donc une fonction pseudo-aléatoire particulière, tirée d'un article (ref [4]), et dont je serait bien incapable de vous décrire le fonctionnement. Sachez que vous pouvez la modifier en remplaçant les nombres premiers de cette formule par d'autres, relativement proches.

Je ne vous guarantis pas que cela soit sans effet sur la répartition des probabilités, ni même la longueur des cycles. L'expérience montre simplement qu'effectivement, substituer d'autres

nombres premiers du même ordre de grandeur offre un résultat tout aussi satisfaisant. Pour plus d'informations à ce sujet, je vous renvoie aux théories mathématiques correspondantes.

Listing 1 – Fonction pseudo-aléatoire

```
//Fournit une valeur aléatoire entre -1 et 1
double rand_noise(int t)
{
        t = (t<<13) ^ t;
        t = (t * (t * t * 15731 + 789221) + 1376312589);
        return 1.0 - (t & 0x7fffffff) / 1073741824.0;
}
//On pourra obtenir une valeur entre 0 et 1 en utilisant
//la formule : (rand_noise(t) + 1.) / 2.
```

2.2 Les fonctions de bruit

Suivant votre choix quant à l'obtention de votre jeu de valeurs aléatoires, l'implémentation de la fonction de bruit diffère.

Dans le cadre où vous travaillez sur une seul dimension – c'est le cas des courbes – votre fonction bruit se limitera à un simple accès à la valeur correspondant à votre coordonnée – que l'on notera x – ou encore à l'appel de la fonction *rand_noise*.

Dans les autres cas, où vous travaillez sur un espace de dimension deux ou supérieure, et que vous avez fait le choix d'utiliser un jeu de valeurs précalculées, il vous faudra générer un tableau bidimensionnel, ou bien encore un simple vecteur que vous utiliserez comme un tableau bidimensionnel.

Listing 2 – Bruit tridimensionnel à partir d'un jeu de valeurs

```
double rand_set[W * H * K];

//Initialisation du tableau, etc...

double noise_3d(int x, int y, int z)
{
        return rand_set[x + y*w + z*w*h];
}
```

Dans le cas contraire, il vous faudra réaliser une fonction de bruit à plusieurs variables. Si vous tentez différentes expressions polynômiales, vous n'obtiendrez rien de bien concluant et des motifs ou déformations se profileront, entraînant la perte du caractère pseudo-aléatoire que vous recherchez. Une bonne solution est la composition de la fonction de bruit. Il est nécessaire de multiplier le résultat de notre fonction de bruit par un scalaire – quelconque, mais de valeur suffisamment "grande" de façon à rendre une faible variation de la fonction de bruit significative – avant chaque composition, puisque nous avons $-1 <= rand\_noise <= 1$. Observez l'exemple suivant :

Listing 3 – Bruit bidimensionnel

```
double noise_2d(int x, int y)
```

```
{
        int tmp = rand_noise(x) * 850000;
        return rand_noise(tmp + y);
}
```

D'où nous déduisons facilement la formule de calcul d'un bruit quadridimensionnel :

Listing 4 – Bruit quadridimensionnel

```
double noise_4d(int x, int y, int z, int t)
{
        int tmp_x = rand_noise(x) * 850000;
        int tmp_y = rand_noise(tmp_x + y) * 850000;
        int tmp_z = rand_noise(tmp_y + z) * 850000;
        return rand_noise(tmp_z + t);
}
```

Et, pour conclure, une formule utilisable pour toute dimension :

Listing 5 – Bruit à n dimensions

```
double noise_nd(int *data_set, int dim)
{
        int i;
        double r = 0.;

        for(i = 0; i < dim; i++)
                r = rand_noise(data_set[i] + (int)(r * 850000) );
        return r;
}
```

3 Interpolations de valeurs

3.1 L'interpolation

L'interpolation est "une opération mathématique permettant de construire une courbe à partir des données d'un nombre fini de points". En d'autres mots, c'est un processus qui consiste à définir une fonction continue prenant certaines valeurs en certains points, et ce selon certains critères que l'on choisit de s'imposer.

La deuxième phase de l'algorithme de Perlin consiste en l'interpolation de valeurs intermédiaires définies régulièrement en certains points de l'espace par une fonction de bruit. Concrètement, imaginons que l'on reprenne notre bruit unidimensionnel de la figure 2 où l'on a associé à chaque entier une valeur pseudo-aléatoire comprise entre -1 et 1, et que l'on souhaite tracer une courbe continue passant par ces points. On souhaite donc définir une fonction $g : \mathbb{R} \to \mathbb{R}$ dont la restriction à $\mathbb{N}$ est f.

Il existe une infinité de fonctions qui respectent ces conditions. Toutefois, nous n'étudierons que trois cas particuliers. Nous ne détaillerons pas les aspects mathématiques et nous nous contenteront d'appréhender, "avec les mains", le fonctionnement de celles ci.

3.2 L'interpolation linéaire

Il s'agit de la solution la plus simple que l'on puisse trouver à ce problème. Puisque nous avont un ensemble de points, pourquoi ne pas se contenter de les relier en traçant des segments qui joignent chaque point à ses deux voisins? On constate alors immédiatement que la fonction est continue et bien définie.

On peut obtenir tous les points du segment AB de façon paramétrique – c'est-à-dire selon une variable t que l'on contrôle – en prenant $t \in [0, 1]$ auquel on associe le point M de coordonnées $(x_A * (1 - t) + t * x_B, y_A * (1 - t) + t * y_B)$. Intuitivement, pour $t = 0$ on se trouve en A, et l'on glisse jusqu'en B de façon linéaire – une autre façon de voir ceci est de constater que l'on se déplace proportionnellement à la progression de t dans l'intervale $[0, 1]$.

Dans notre cas, nous souhaitons obtenir l'ordonnée en fonction de l'abscisse. Or, si l'on définit les points dont la valeur – issue de notre bruit – est imposée parmi les entiers, alors la partie décimale de chaque coordonnée correspond à notre paramètre.

On prendra donc la partie fractionnaire de x comme valeur de t.

Ce qui nous amène finalement à la fonction d'interpolation suivante :

Listing 6 – Interpolation linéaire

```
1 double linear_interpolate(double a, double b, double t)
  {
          return (1. - t) * a + t * b;
  }
```

3.3 L'interpolation cosinusoïdale

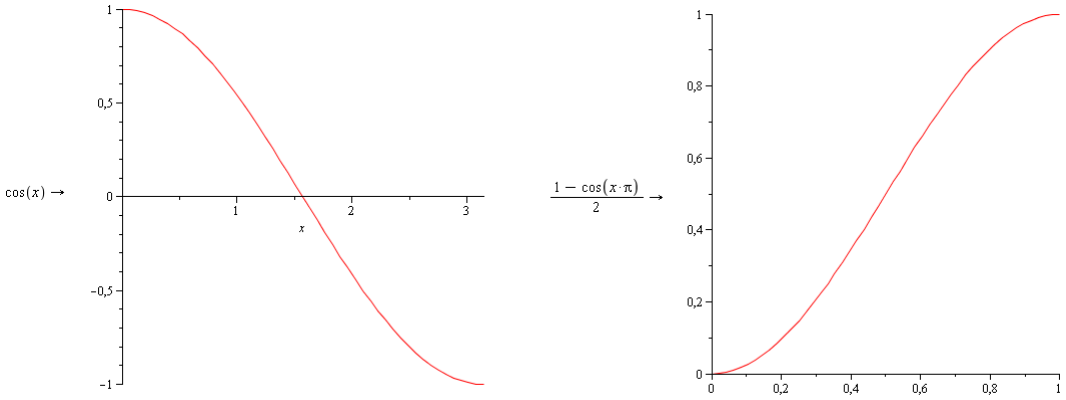
L'un des problèmes de l'interpolation linéaire est son aspect irrégulier, ce qui se traduit mathématiquement par l'abscence d'une dérivée première. La condition que l'on va alors imposer est que la fonction obtenue soit continue et dérivable, à dérivée continue. On dira qu'elle est de classe C^1, alors que l'interpolation linéaire est de classe C^0.

Puisque le problème de l'interpolation linéaire se situe aux jonctions des segments, nous allons leur substituer une courbe dont la dérivée s'annule à chaque extrémité, nous assurant ainsi la dérivabilité.

Une fonction qui se prête bien à ceci est bien sûr la fonction cosinus, dont la dérivée en 0 et en $\frac{\pi}{2}$ s'annule. En appliquant une légère transformation, nous pouvons définir la fonction $c : [0,1] \to [0,1] x \mapsto \frac{1-cos(x)}{2}$ [1].

Vous trouvez les courbes de ces deux fonctions à la figure 3.

FIGURE 3 – La fonction cosinus et notre fonction particulière



Se pose alors la question de "déformer" cette fonction pour que chacune de ses extrémitées correspondent aux deux points que l'on souhaite joindre. En fait, plutôt que de chercher à déformer notre fonction, cherchons plutot comment transformer notre interpolation linéaire. On observe que si l'on substitue t par $c(t)$ on obtient bien une fonction dont la dérivée est nulle en 0 et 1, ce que nous recherchons.

Concrètement, cela se traduit par une contraction des distances au voisinage des extrémités, et à l'inverse par une élongation de celles-ci vers le centre de notre arc de courbe.

Une autre façon, plus cinématique, de se représenter la chose est de percevoir que le déplacement selon t est faible aux extrémités des segments et extrêmement rapide au voisinage de $\frac{1}{2}$. Ce que confirme le graphe de la dérivée de c (cf figure 4).

Ce qui nous donne, finalement l'extrait 7

1. Il existe des fonctions polynômiales qui ont une courbe et des propriétés de dérivabilité similaires, par exemple la fonction polynômiale d'Hermite. Nous en parlons dans la partie avancée de ce document.

Listing 7 – Interpolation cosinusoïdale

```
1 double cosine_interpolate(double a, double b, double t)
  {
          double c = (1 - cos(t * 3.1415927)) * .5;

          return (1. - c) * a + c * b;
6 }
```

Le rendu est particulièrement plus doux, comme le montre la figure 5. Persistent toutefois quelques aberrations – des exemples se trouvent aux centres des cercles – qui ne correspondent pas vraiment à l'idée que l'on se ferait de cette fonction, ce qui justifiera l'introduction d'une troisième méthode, plus lourde et plus complexe, mais au résultat visuellement plus agréable.

FIGURE 4 – La dérivée de notre fonction c

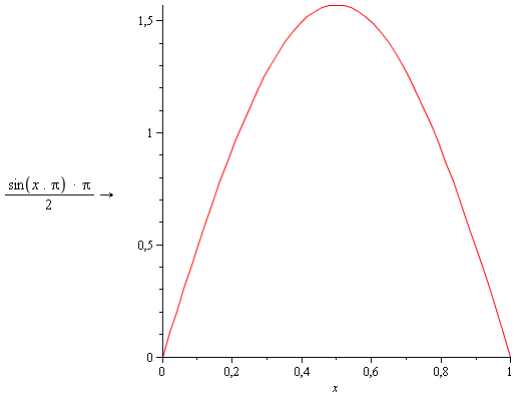
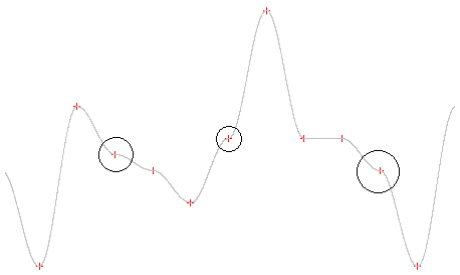


FIGURE 5 – Interpolation cosinusoïdale



3.4 L'interpolation cubique

L'interpolation cubique apporte une solution aux problèmes que nous avons décelés alors que nous nous intéressions à l'interpolation cosinusoïdale. Nous allons utiliser des polynômes du troisième degré pour obtenir une approximation de la fonction en deux points.

En effectuant un recollement par morceaux, tout comme nous l'avons fait avec l'interpolation cosinusoïdale, nous obtiendrons une fonction de classe C^2, c'est-à-dire deux fois dérivable à dérivée continue. Cela traduit un critère de régularité plus important.

Non seulement la courbe doit être lisse, mais les tangentes à cette courbe doivent varier de façon continue. C'est cependant une méthode coûteuse puisqu'elle ne nécessitera non pas deux mais quatre points.

Si nous voulons ajouter la continuité de la dérivée seconde en chacun des points, il devient nécessaire de faire intervenir les points situés avant et après la portion de courbe que nous souhaitons interpoler.

Nous nous retrouvons alors avec quatre équations – vous pouvez obtenir plus de détails en consultant la bibliographie. Pour respecter ces quatre contraintes, il faut disposer de quatre variables que nous pouvons contrôler, ce qui nous amène à choisir un polynôme de degré trois.

Bien que l'idée soit de calculer la dérivée seconde en fonction des points précédents, il existe diverses façons de "règler" nos coefficients.

Nous allons utiliser un cas particulier (extrait 8). Il nous permettra d'obtenir une fonction polynômiale qui, définie sur l'intervale $[-1, 1]$ a pour avantage de ne prendre que rarement ses valeurs hors de celui-ci.

Listing 8 – Inteprolation cubique

```
//Interpolation des valeurs situées entre p0 et p1
//Nécessite deux points qui précèdent (resp. succèdent)
// à p1 (rep. p2).
double cubic_interpolate(double before_p0, double p0,
                         double p1, double after_p1)
{
        //Calcul des coefficients de notre polynôme
        double a3 = -0.5*before_p0 + 1.5*p0 - 1.5*p1 + 0.5*after_p1;
        double a2 = before_p0 - 2.5*p0 + 2*p1 - 0.5*after_p1;
        double a1 = -0.5*before_p0 + 0.5*p1;
        double a0 = p0;

        //Calcul de la valeur de ce polynôme en t
        return (c3 * t*t*t) + (c2 * t*t) + (c1 * t) + c0;
}
```

La différence est tout de suite perceptible, comme le montre la figure 6.

3.5 Interpolation du bruit

Nous disposons maintenant de fonctions qui nous permettent d'interpoler entre deux valeurs de notre bruit. Nous pouvons donc écrire une fonction de "bruit lissé". Elle prend donc en para-

mètre la coordonnée x, dont elle sépare la partie entière et la partie fractionnaire, pour ensuite interpoler entre le point de coordonnée (x) et celui de coordonnée $(x+1)$.

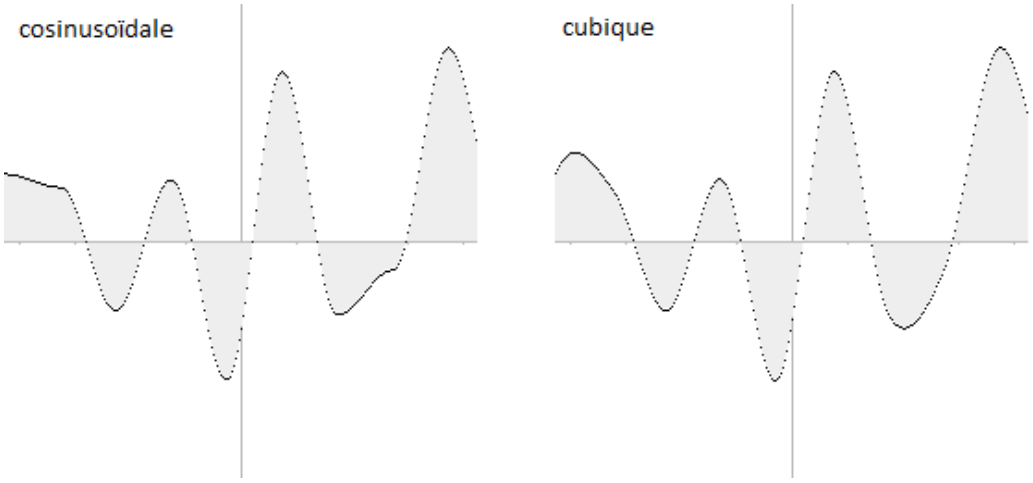
Dans un premier temps, le cas linéaire :

Listing 9 – Bruit lissé

```
double smooth_noise(double x)
{
        //Partie entière : E(x)
        int integer_x = (int)x;
        //Partie fractionnaire : x - E(x)
        double fractional_x = x - integer_x;


        //Bruit du point précédent :
        double a = noise(integer_x);
        //Bruit du point suivant :
        double b = noise(integer_x + 1);

        //Interpolation :
        return linear_interpolate(a, b, fractional_x);
}
```

Pour une interpolation cosinusoïdale nous avons la même chose, à la fonction d'interpolation près. Le cas de l'interpolation cubique est légèrement plus complexe puisqu'il nécessite quatre points.

FIGURE 6 – Interpolation cosinusoïdale <-> Interpolation cubique



Listing 10 – Bruit lissé

```
double smooth_noise(double x)
{
        //Partie entière : E(x)
        int integer_x = (int)x;
        //Partie fractionnaire : x - E(x)
        double fractional_x = x - integer_x;


        //Bruit des quatre points
        double c = noise(integer_x - 1);
        double a = noise(integer_x);
        double b = noise(integer_x + 1);
        double d = noise(integer_x + 2);

        //Interpolation :
        return cubic_interpolate(c, a, b, d, fractional_x);
}
```

3.6 Vers une autre dimension

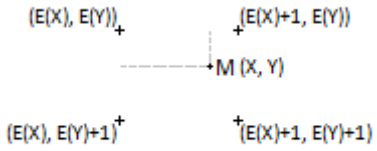
Nous avons découvert trois méthodes, dans un ordre de coût, en temps de calcul, croissant. Malheureusement, toutes ces méthodes se sont limitées à une interpolation monodimensionnelle.

Qu'en est-il du cas, plus probable, où nous nous retrouvons avec deux, trois, ou même quatre dimensions – ce qui est plus courant que l'on pourrait le penser ; si l'on souhaite appliquer une texture de bruit de Perlin à un objet tridimensionnel, et ceci tout en animant la texture, on aurait alors besoin d'une quatrième dimension qui serait ici le temps.

Il existe une méthode qui permet de généraliser chacune de celles-ci à une dimension n, et c'est celle que nous allons décrire.

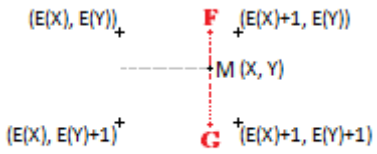
Supposons, dans un premier temps, que nous souhaitons interpoler un bruit bidimensionnel. Nous raisonnons donc dans le plan. Si l'on conserve notre habitude de placer nos points aux coordonnées entières, nous souhaitons donc interpoler la valeur d'un point M qui se trouve environné de quatre autres points, et sa valeur sera donc dépendante de chacun de ces points. Si l'on note (X, Y) les coordonnées de M, et que $E(z)$ représente la partie entière de z, les 4 points ont pour coordonnées : $A = (E(X), E(Y))$, $B = (E(X) + 1, E(Y))$, $C = (E(X), E(Y) + 1)$, $D = (E(X) + 1, E(Y) + 1)$

FIGURE 7 – Un point dans notre plan



Cherchons à subdiviser le problème pour le rendre plus simple. Nos quatre points forment deux à deux, le long de l'axe Ox, des segments. Nous pouvons donc chercher à interpoler le long du segment $[AB]$ en prenant la partie fractionnaire de X comme troisième paramètre de notre fonction d'interpolation. Nous pouvons faire de même pour le segment $[CD]$. Nous nous retrouvons alors avec les deux valeurs des points F et G de la figure 8.

FIGURE 8 – Etape intermédiaire de l'interpolation linéaire dans le plan



Nous pouvons, une troisième fois, interpoler le long du segment $[FG]$ afin d'obtenir la valeur en M. Pour cela, nous prendrons les deux valeurs calculées précédemment ainsi que la partie fractionnaire de la coordonnée Y de M.

Si l'on résume le code correspondant, nous obtenons :

Listing 11 – Interpolation linéaire 2D

```
//...

double f = linear_interpolate(a, b, fractional_x);
double g = linear_interpolate(c, d, fractional_x);

double result = linear_interpolate(f, g, fractional_y);
```

Nous pouvons généraliser ceci à trois ou quatres dimensions. Pour n dimensions, il suffit d'interpoler les n-1 dimensions deux fois, avant d'interpoler les deux valeurs résultantes.

Un exemple en dimension trois:

Listing 12 – Inteprolation linéaire 3D

```
double smooth_noise(double x, double y, double z)
{
  //Partie entière : E(x)
  int integer_x = (int)x;
  int integer_y = (int)y;
  int integer_z = (int)z;
  //Partie fractionnaire : x - E(x)
  double fractional_x = x - integer_x;
  double fractional_y = y - integer_y;
  double fractional_z = z - integer_z;

  //Bruit des quatre points d'un cube
  double a0 = noise(integer_x,     integer_y,     integer_z);
  double a1 = noise(integer_x + 1, integer_y,     integer_z);

  double b0 = noise(integer_x,     integer_y + 1, integer_z);
  double b1 = noise(integer_x + 1, integer_y + 1, integer_z);

  double c0 = noise(integer_x,     integer_y,     integer_z + 1);
  double c1 = noise(integer_x + 1, integer_y,     integer_z + 1);

  double d0 = noise(integer_x,     integer_y + 1, integer_z + 1);
  double d1 = noise(integer_x + 1, integer_y + 1, integer_z + 1);

  //Interpolation sur la face inférieure du cube :
  double a = linear_interpolate(a0, a1, fractional_x);
  double b = linear_interpolate(b0, b1, fractional_x);
  double v = linear_interpolate(a, b, fractional_y);
  //Interpolation sur la face supérieure du cube :
  double c = linear_interpolate(c0, c1, fractional_x);
```

```
  double d = linear_interpolate(d0, d1, fractional_x);
  double w = linear_interpolate(c, d, fractional_y);

  //Interpolation entre les points
  // situés sur chacune des deux faces :
  return linear_interpolate(v, w, fractional_z);
}
```

Il est évident que pour des dimensions plus élevées, nous n'allons pas expliciter le calcul pour chacun de nos points. Nous préférerons une méthode récursive. L'extrait 13 est un exemple valable pour toute dimension. Il reste toutefois délicat pour une première lecture.

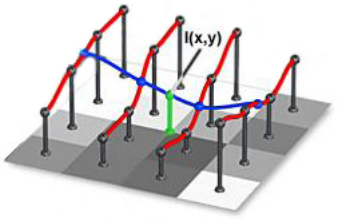
Listing 13 – Inteprolation cosinusoïdale nD

```
double smooth_noise(double data[], int dim)
{
        return _smooth_noise(data, dim, dim);
}

double _smooth_noise(double data[], int dim, int dim_work)
{
        //Condition d'arrêt de la boucle récursive
        //Nous permet d'obtenir les points
        if(dim_work <= 0)
                //Fonction de bruit multidimensionnel
                return noise(data, dim);

        //Récupère la dernière dimension sur
        // laquelle nous travaillons
        double x = data[dim_work - 1];
        int integer_x = (int)x;
        double fractional_x = x - integer_x;

        //Interpolation de la dimension dim_work - 1,
        // avec x = integer_x
        data[dim_work - 1] = integer_x;
        double a = _smooth_noise(data, dim, dim_work - 1);

        //Interpolation de la dimension dim_work - 1,
        // avec x = integer_x + 1
        data[dim_work - 1] = integer_x + 1;
        double b = _smooth_noise(data, dim, dim_work - 1);

        //Restauration du tableau, pour ne pas
        //perdre la valeur en sortant de la fonction
        data[dim_work - 1] = x;

        //Interpolation de la dimension dim_work
        return cosine_interpolate(a, b, fractional_x);
}
```

3.7 Splines cubiques

Nous avons pu généraliser les interpolations linéaires et cosinusoïdales à plusieurs dimensions. Ceci peut aussi s'appliquer à notre interpolation cubique. Vous pourrez en entendre parler sous le nom de "spline cubique". Cette fois, nous ne nous contenterons pas de deux points, mais de quatres points. Le plus simple reste encore un schéma (figure 9), pour la dimension deux.

FIGURE 9 – Splines cubiques



Le mécanisme est identique aux exemples précédent, à ceci près que nous nécessiterons quatre points par inteprolation. Nous nous contenterons d'un exemple en dimension deux.

Listing 14 – Inteprolation cubique 2D

```
//Nous interpolons sur une ligne, pour un y fixé
double smooth_noise_firstdim(int integer_x,
                             int integer_y, double fractional_x)
{
        double v0 = noise(integer_x - 1, integer_y);
        double v1 = noise(integer_x,     integer_y);
        double v2 = noise(integer_x + 1, integer_y);
        double v3 = noise(integer_x + 2, integer_y);

        return cubic_interpolate(v0, v1, v2, v3, fractional_x);
}

//Nous interpolons sur les y, en utilisant la fonction précédente
double smooth_noise(double x, double y)
{
        int integer_x = (int)x;
        double fractional_x = x - integer_x;
        int integer_y = (int)y;
        double fractional_y = y - integer_y;

        double t0 = smooth_noise_firstdim(integer_x,
                                          integer_y - 1, fractional_x);
        double t1 = smooth_noise_firstdim(integer_x,
                                          integer_y,     fractional_x);
        double t2 = smooth_noise_firstdim(integer_x,
```

```
                                                integer_y + 1, fractional_x);
        double t3 = smooth_noise_firstdim(integer_x,
                                                integer_y + 2, fractional_x);
29
        return cubic_interpolate(t0, t1, t2, t3, fractional_y));
}
```

4 Le bruit de Perlin

4.1 Compréhension et implémentation

Nous diposons maintenant de tous les outils nécessaires à la réalisation de notre “bruit de Perlin”.

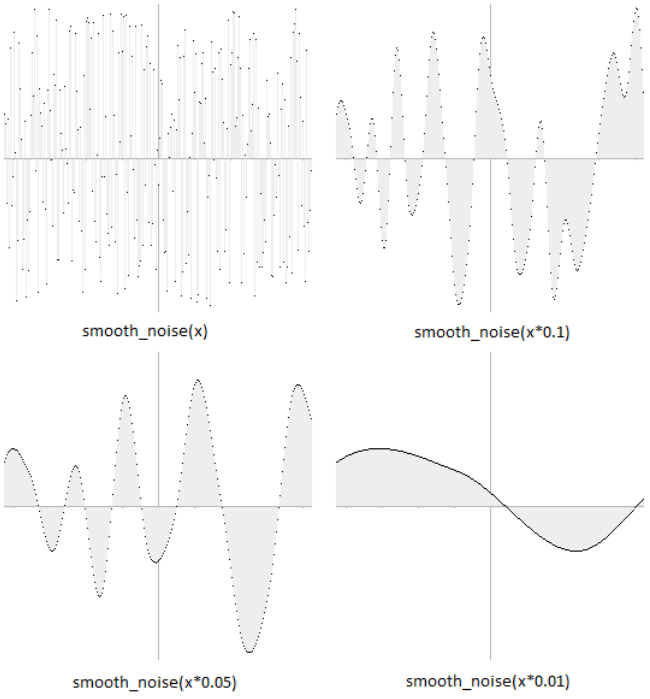
Le bruit de Perlin est formé d’une somme de fonctions de bruit, dont la *fréquence* et l’*amplitude* varie. Ici, nous appellerons *fréquence* l’inverse du *pas*, et nous appellerons *pas* l’intervale entre deux points définis par notre bruit. Jusqu’à présent, deux de nos points étaient séparés par une distance de 1, mais nous aurions très bien pu choisir 10, ou bien 0.5.

Nous allons donc chercher à faire varier le pas. Pour ce faire, nul besoin de modifier notre fonction *smooth_ noise* ; si nous multiplions nos coordonnées par 2, avant d’appeler cette fonction, nous divisons l’intervale entre deux points par deux. Si, à l’inverse, nous les multiplions par "0.5", alors nous multiplions l’intervale entre deux points par deux.

Si nous cherchons donc à multiplier notre pas par k, nous multiplions nos coordonnées par la fréquence $\frac{1}{k}$.

La figure 10 représente différents appels à notre fonction, où le paramètre x de notre fonction de “bruit lissé” est la coordonnée x du pixel de l’image.

FIGURE 10 – Effet de quelques fréquences sur *smooth_ noise*

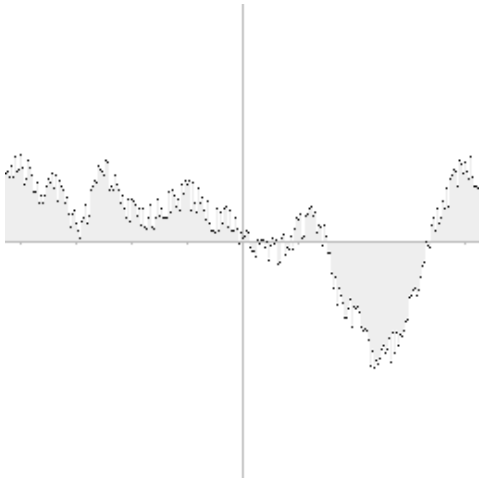


Nous allons donc sommer des courbes d’amplitude de plus en plus faible, le contrôle de la variation d’amplitude de chacune de ces courbes – et donc, leur part dans la courbe finale – lors

de cette somme se fera par un paramètre que l'on nommera la persistance. Ceci, adjoint à des courbes de variation de plus en plus rapide, va créer l'effet "nuageux" du bruit de Perlin. Pour être exacts, nous approximons une fractale où si l'on observe de plus près une zone particulière, nous retrouvons le même motif qu'à l'échelle précédente.

Si nous reprenons nos courbes de l'exemple précédent et que nous les sommons, en pondérant l'amplitude de chacune de ces fonctions, nous obtenons la figure 11.

FIGURE 11 – Somme des fréquences 1, 0.1, 0.05 et 0.01 avec une persistance de 0.5



Nous allons donc réaliser une fonction qui prendra en argument :
— Le nombre d'octaves n : le nombre d'appels à la fonction de "bruit lissé"
— La fréquence f: la fréquence de la première fonction de "bruit lissé"
— La persistance p: correspond au facteur qui vient modifier l'amplitude de chaque fonction de "bruit lissé"

Concrètement, nous allons faire la somme de n appelles à *smooth_noise* en multipliant à chaque fois la fréquence par deux, et l'amplitude (qui vaut initialement 1) par la persistance. On obtiendra donc la fonction : $f : x \mapsto \Sigma_{i=0}^{n-1} p^i * smooth\_noise(f * 2^i * x)$.

Attention, pour conserver nos valeurs dans l'intervalle $[-1, 1]$, nous devons diviser le résultat total par la somme des amplitudes. Pour simplifier nos calculs et éviter de sommer chacune d'elle, on peut utiliser la formule de la somme des termes d'une série géométrique $\sum_{i=0}^{n-1} p^i = \frac{1-p^n}{1-p}$ – à condition que $p \neq 1$. On divisera donc par cette valeur.

Nous parvenons donc, en termes de code C, à l'extrait 15.

Listing 15 – Bruit de Perlin 1D

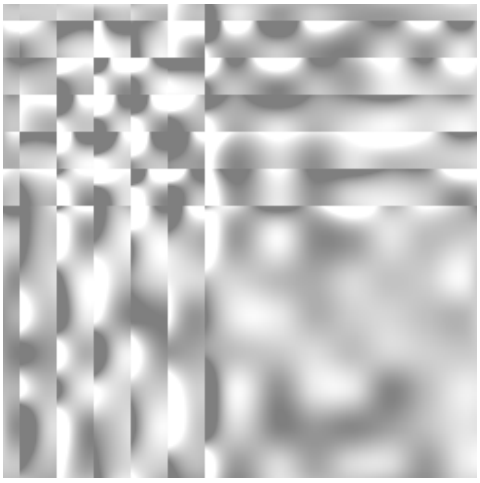
```
double perlin(int octaves, double frequency,
              double persistence, double x)
{
    double r = 0.;
    double f = frequency;
```

```
        double amplitude = 1.;

        for(int i = 0; i < octaves; i++)
        {
                r += smooth_noise(x * f) * amplitude;
                amplitude *= persistence;
                f *= 2;
        }

        double geo_lim = (1 - persistence) / (1 - amplitude);

        return r * geo_lim;
}
```

4.2 Pixelisation aux coordonnées négatives

Si vous vous contentez de la version proposée, et que vous souhaitez utiliser l'une des fonctions de "bruit lissé" avec des coordonnées négatives, vous vous apercevrez alors que nos valeurs sont incorrectes. Vous obtiendrez probablement quelque chose de comparable à la figure 12.

FIGURE 12 – Apparition d'erreurs pour des coordonnés négatives



Le problème vient de notre façon de récupérer la partie décimale pour des valeurs négatives. En effet, convertir -0.5 en entier nous donne 0 là où nous attendrions -1.

Il faut donc corriger avec un bloc conditionnel similaire à l'extrait 16.

Listing 16 – Correction de l'erreur pour la partie entière

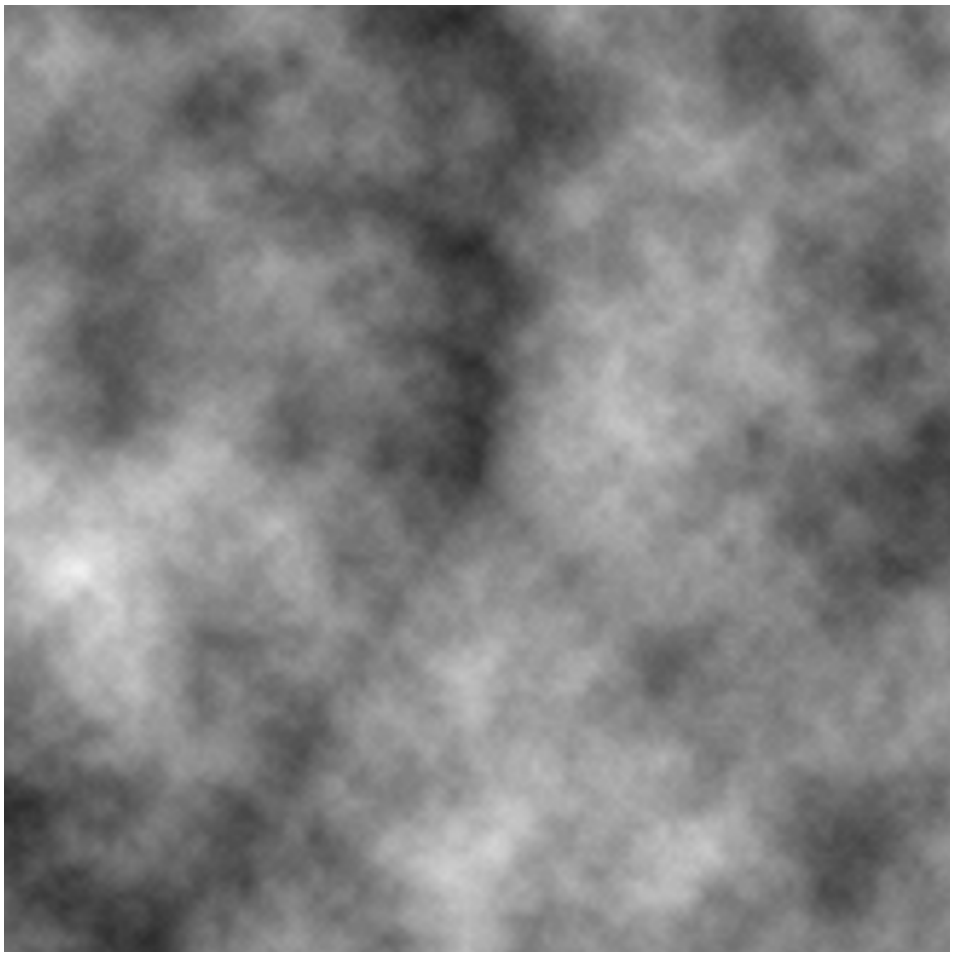
```
if(x >= 0)
        integer_x = (int)x;
```

```
else
        integer_x = (int)x - 1;
fractional_x = x - integer_x;
```

4.3 Motif fractal évident au centre d'homothétie

Ce premier problème corrigé, intéressons-nous au bruit de Perlin, toujours au centre de notre repère. Un regard attentif saura discerner la redondance du même motif, dont seule la taille varie. Vous pourrez même, avec un peu de chance, compter combien d'exemplaires du même nuage vous retrouvez, tous alignés sur une droite passant par l'origine (cf : figure 13). Vous remarquerez alors que cela correspond exactement à l'octave de votre fonction de "bruit de Perlin".

FIGURE 13 – Apparition de droites passant par l'origine



Ce problème est dû au simple fait que chacune de vos fonctions de bruit ont pour centre

d'homothétie l'origine. Ceci peut facilement se régler en ajoutant une *translation* du centre de cette homothétie qui est *fonction de l'indice de la fonction de bruit.*

Plutôt que de longs discours, un exemple concret sera bien plus parlant.

Listing 17 – Ajout d'une translation fonction de l'octave au bruit de Perlin 1D

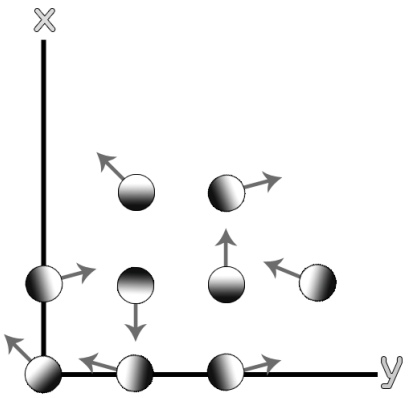
```
double perlin(int octaves, double frequency,
              double persistence, double x)
{
        double r = 0.;
        double f = frequency;
        double amplitude = 1.;

        for(int i = 0; i < octaves; i++)
        {
                //Translation du centre de symétrie en i * 4096
                int t = i * 4096;

                //Calcul du bruit translaté
                r += smooth_noise(x * f + t) * amplitude;

                amplitude *= persistence;
                f *= 2;
        }

        double geo_lim = (1 - persistence) / (1 - amplitude);

        return r * geo_lim;
}
```

5 L'algorithme original

5.1 Champs vectoriels

L'algorithme que je vous ai présenté jusqu'ici est le plus simple, mais de loin le moins efficace, et diffère du bruit de Perlin original, tel que K. Perlin l'implémenta. Jusqu'ici nous avons considéré un champ scalaire, c'est-à-dire qu'à chaque point de notre espace, nous avons associé une valeur scalaire comprise dans l'intervale $[-1, 1]$. L'approche de l'alrithme original est légèrement différente. En effet, plutot que d'associer à chaque point entier de notre espace une valeur, nous allons lui associer un vecteur, qui définira un gradient de couleur. Ce gradient, représente une variation de la valeur -1 à la valeur 1.

FIGURE 14 – Champ de gradient



Par la suite, nous effectuerons le produit scalaire de chacun de ces vecteurs de gradient, avec le vecteur allant du point auquel est associé ce gradient vers le point de l'espace que nous considérons. Nous obtiendrons ainsi des valeurs scalaires – 4 pour un plan – que nous combinerons en utilisant une interpolation similaire à l'interpolation cosinusoïdale.

À l'origine, K. Perlin utilisa la fonction polynômiale d'Hermite, à savoir $f(t) = 3t^2 - 2t^3$ qui présente une courbe tout à fait adaptée à ce que nous cherchons (Cf: Interpolation cosinusoidal) et qui vérifie bien la condition d'annulation de ses dérivées premières en 0 et 1. Toutefois, il est plus que souhaitable que les dérivées secondes s'annulent aussi en ces points, assurant ainsi une "meilleure" continuité[2]. Ceci peut être obtenu grâce à la fonction polynômiale $f(t) = 6t^5 - 15t^4 + 10t^3$.

5.2 Tables de hachage

Afin d'accroître de façon considérable la vitesse de calcul du bruit (sans perte de qualité notable) nous utiliserons des tables de hachage pour les deux opérations essentielles de notre calcul.

2. Les effets sont visible lorsque l'on utilise le bruit de Perlin pour du bump mapping, ou encore de la génération de terrain. En effet, ce sont ici les dérivées premières et secondes qui entrent en jeu. La non utilisation de ce polynôme provoque des discontinuités, parfois très visibles. Je vous renvoie au document [12].

Dans un premier temps, nous n'utiliserons plus notre fonction de bruit aléatoire, mais une permutation $\sigma(n)$ sur l'ensemble des 256 premiers entiers. Cela signifie qu'à chaque nombre compris entre 0 et 255, nous lui associons une nouvelle valeur entière $\sigma(n)$ comprise 0 à 255. Cette table de permutation remplacera notre fonction pseudo-alléatoire. Il va de soi qu'elle doit être, elle-même, pseudo-alléatoire. Nous prendrons la table proposée dans un des documents de référence, qui convient parfaitement.

Puisque nos valeurs sont théoriquements infinies, nous devrons ramener la partie entière de chacune des coordonnées à l'intervale $[0, 255]$. Comme ces valeurs sont judicieusement choisies, pour calculer le modulo il nous suffira de conserver les 8 premiers bits de chacune des coordonnées[3], ce qui est une opération élémentaire des plus rapides.

Nous composerons nos permutations sur le même modèle que la génération de bruit multidimensionnel. Nous aurons donc $\sigma((\sigma((\sigma(X\%256) + Y)\%256) + Z)\%256)$.

Afin, toujours, de réduire le nombre d'opérations, nous utiliserons une table qui fait corres-

3. Pour rappel, un calcul de modulo possède un coût identique à celui d'une division. Aussi, puisqu'une division par une puissance de deux est un simple décalage binaire, le calcul d'un modulo par une puissance de deux est une simple conservation de bits.

FIGURE 15 – Vecteurs à considérer

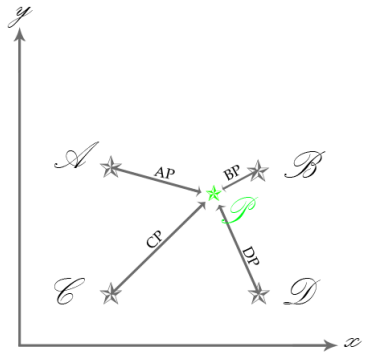
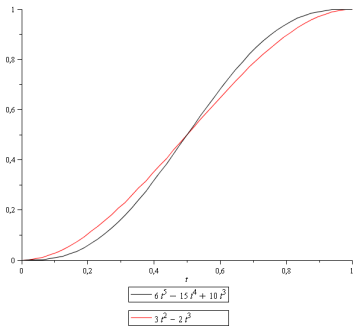


FIGURE 16 – Fonctions polynômiales



pondre les 512 premiers entiers aux 256 premiers. Nous la construirons à partir de la première, et poserons perm[i] = sigma[i & 0xFF] pour tout i de 0 à 511.

Dans un second temps, nous formerons une table de hachage qui fera correspondre à notre valeur aléatoire un vecteur. Nous choisirons ces vecteurs de façon à ce que leur norme soit du même ordre de grandeur, et qu'ils soient approximativement également répartis dans l'espace. Concrètement, pour un bruit 2D nous prendrons des points répartis un cercle, et pour un bruit 3D le milieu des arêtes d'un cube.

Il n'est pas nécessaire de posséder 255 vecteurs, et l'outil mathématique qu'est le modulo nous sera très utile. Seuls 12 vecteurs surfisent pour un bruit 3D. Mais afin de faciliter le calcul du modulo, nous pourons prendre 16 vecteurs, en ajoutant simplement le tétraèdre régulier formé des points $(1,1,0),(-1,1,0),(0,-1,1),(0,-1,-1)$.

5.3 Une implémentation du cas tridimensionnel

Pour décrire cette algorithme de façon efficace, je vous propose un code commenté (extrait 18), reprenant ce dont nous avons parlé précédemment, correspondant a l'implémentation d'un bruit de Perlin tridimensionnel. Vous pourrez facilement l'adapter en une version bidimensionnel, ou bien ajouter quelques vecteurs pour former un hypercube et obtenir un bruit 4D [4].

Listing 18 – Implémentation d'un bruit de Perlin tridimensionnel

```
//////
//La table de permutation :
//////
//Elle associe à chaque valeur comprise entre 0 et 256 une unique
// valeur elle aussi comprise entre 0 et 256. C'est une permutation.
//Afin d'éviter des opérations de modulo, dans le souci d'accroître
// la vitesse de calcul, elle est définie de 0 à 512.
unsigned char perm[512] = {
        //0 à 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,
        //257 à 512
        151,160,137,91,90,15,131,13,201,95,96,53,194,233,7,225,140,
        //...
        127,4,150,254,138,236,205,93,222,114,67,29,24,72,243,141,
        128,195,78,66,215,61,156,180};

//////
//La table des vecteurs :
//////
//Elle contient les gradients de couleur
// que nous allons associer à chaque point de l'espace.
static int _grad3[16][3] = {
        //Le cube
```

4. K. Perlin conseil, dès que l'on a affaire à plus de trois dimensions, d'utiliser le "simplex noise", dont il est question dans le chapitre suivant. Si l'on compare la complexitée logarithmique du *bruit de Perlin* et du *simplex noise* relativement à la dimension, le premier est en $O(2^n)$ contre n^2 pour le second.

```
        {1,1,0},{-1,1,0},{1,-1,0},{-1,-1,0},
        {1,0,1},{-1,0,1},{1,0,-1},{-1,0,-1},
        {0,1,1},{0,-1,1},{0,1,-1},{0,-1,-1},
        //Ainsi qu'un tétraèdre supplémentaire
        {1,1,0},{-1,1,0},{0,-1,1},{0,-1,-1}};

////
//Fonction : Produit scalaire
////
//Effectue le produit scalaire du vecteur V(v[0], v[1], v[2])
// avec U(x, y, z).
//Cette fonction nous servira à calculer le produit scalaire
// entre nos gradients de couleur et nos vecteurs dirigés
// vers notre point.
double fast_dot(const int *v, const double x,
                const double y, const double z)
{
        return v[0] * x + v[1] * y + v[2] * z;
}

////
//Fonction : Obtention du gradient pour un point P(x, y, z)
//             de l'espace
////
int *get_grad(int x, int y, int z)
{
        //Calcul un bruit aléatoire de 3 variables, via la table
        // de permutation.
        int rand_value = perm[z + perm[y + perm[x]]];
        //Applique un modulo 16 à cette valeur pour obtenir un
        // gradient de couleur, puis renvoie un pointeur sur
        // cette élément.
        return _grad3[rand_value & 15];
}

////
//Fonction : Fonction polynômiale à dérivées première
//             et seconde nulles
////
//Calcule simplement la valeur de ce polynôme en x=t
double quintic_poly(const double t)
{
        const double t3 = t * t * t;
        return t3 * (t * (t * 6. - 15.) + 10.);
}

////
//Fonction : Sépare la partie entière et fractionnaire
////
```

```
void int_and_frac(double value, int *integer_part,
                          double *fractional_part)
{
        integer_part = (int)value;
        if(value < 0)
                integer_part -= 1;
        fractional_part = value - integer_part;
}

//La fonction principale, permettant d'obtenir le bruit lissé
double smooth_noise_3d(double x_pos, double y_pos, double z_pos)
{
        //Les parties entières
        int X, Y, Z;
        //Les parties fractionnaires
        // x, y, z;

        //Comme pour le précédent algorithme, nous séparons parties
        // entière et fractionnaire.
        int_and_frac(x_pos, &X, &x);
        int_and_frac(y_pos, &Y, &y);
        int_and_frac(z_pos, &Z, &z);

        //Nous appliquons un modulo 256, de façon à ce que ces
        // valeurs soient comprises dans notre table de permutation,
        // et que nous puissions utiliser la fonction d'obtention
        // de gradient.
        X &= 255;
        Y &= 255;
        Z &= 255;

        //Nous récupérons les gradient en chacun des sommets du cube
        // contenant notre point.
        //Nous faisons alors le produit de chacun de ces gradients
        // obtenu en un sommet S par le vecteur issu de S et dirigé
        // vers notre point M(x_pos, y_pos, z_pos).
        //On retrouve facilement ces valeurs
        // en dessinant un schéma de notre cube.
        //Chacune de ces variables (résultat de ce produit scalaire)
        // porte un nom constitué de la lettre 'g' suivie des
        // coordonnées x, y, et z du point du cube dont il est issu.
        const double g000 = fast_dot(get_grad(X, Y, Z),
                x, y, z);
        const double g001 = fast_dot(get_grad(X, Y, Z + 1),
                x, y, z - 1.);
        const double g010 = fast_dot(get_grad(X, Y + 1, Z),
                x, y - 1., z);
        const double g011 = fast_dot(get_grad(X, Y + 1, Z + 1),
                x, y - 1., z - 1.);
```

```
        const double g100 = fast_dot(get_grad(X + 1, Y, Z),
                x - 1., y, z);
        const double g101 = fast_dot(get_grad(X + 1, Y, Z + 1),
                x - 1., y, z - 1.);
        const double g110 = fast_dot(get_grad(X + 1, Y + 1, Z),
                x - 1., y - 1., z);
        const double g111 = fast_dot(get_grad(X + 1, Y + 1, Z + 1),
                x - 1., y - 1., z - 1.);

        //Comme pour l'interpolation cosinusoïdale, nous calculons
        // le polynôme pour chacune de nos valeurs d'interpolation :
        // u pour l'interpolation le long de l'axe des x
        // v pour l'interpolation le long de l'axe des y
        // w pour l'interpolation le long de l'axe des z
        const double u = quintic_poly(x);
        const double v = quintic_poly(y);
        const double w = quintic_poly(z);
        //Comme nous l'avons fait avec l'inteprolation cubique,
        // nous composerons :
        // l'interpolation le long de l'axe x par
        // l'inteprolation le long de l'axe y.

        //Nous interpolons le long de l'axe des x sur chacune
        // des arètes parallèles à cet axe de notre cube.
        const double x00 = Math::linear_interpolate(g000 , g100, u);
        const double x10 = Math::linear_interpolate(g010 , g110, u);
        const double x01 = Math::linear_interpolate(g001 , g101, u);
        const double x11 = Math::linear_interpolate(g011 , g111, u);

        //Nous interpolons les arètes deux à deux parallèles
        // se trouvant sur la même face de notre cube.
        const double xy0 = Math::linear_interpolate(x00 , x10, v);
        const double xy1 = Math::linear_interpolate(x01 , x11, v);

        //Enfin, nous interpolons entre les faces inférieur et
        // la face supérieur de notre cube.
        const double xyz = Math::linear_interpolate(xy0 , xy1, w);

        return xyz;
}

//La somme des diverses octaves pour obtenir le motif nuageux
// est identique au précédent algorithme.
```

6 Simplex Noise

Si l'on souhaite disposer d'un bruit de perlin quadrimensionel, ou de dimension supérieure, on se retrouve face à un temps de calcul conséquent que l'on souhaiterait réduire. Pour parer à cette difficulté, Ken Perlin proposa le *Simplex noise*. Le terme *Simplex* fait références aux simplexes réguliers, les solides réguliers que l'on peut construire avec un nombre minimal de points à dimension n fixé. En dimension deux, nous trouvons le triangle équilatéral, et en dimension trois, le thétraèdre.

6.1 Principe

L'idée essentielle est de paver l'espace de dimension n avec des simplexes réguliers de coté 1. À chaque sommet s issu de ce pavage, on associe un vecteur gradient $\vec{G_s}$, tout comme nous l'avions fait dans le cas du *bruit de Perlin*. On donne alors les coordonnées d'un point $P(x_1, \ldots, x_n)$ et l'on cherche les trois sommets les plus proches, c'est-à-dire à quel n-simplexe ce point appartient. Si l'on note $A_1, \ldots, A_{n+1}$ les $n+1$ sommets de ce simplexe, on détermine alors les vecteurs $\vec{A_1P}, \vec{A_2P}, \ldots, \vec{A_{n+1}P}$. Enfin, on choisit une fonction radiale $f(r)$ qui s'annule et change de signe si la distance $r = |\vec{A_iP}|$ de P à l'un des sommets A_i est supérieure [5] à $\frac{1}{2}$. La valeur du bruit en P est alors la somme, sur chacun des sommets ;

$$\sum_{i=0}^{n+1} f(r_i)\vec{A_iP}.\vec{G_{A_i}} \text{ où } r_i = |\vec{A_iP}|$$

C'est-à-dire la somme des produits scalaire des vecteurs gradient par les vecteurs "sommet→point", pondérée par les fonctions radiales. La figure 17 présente l'application de cette formule à un unique sommet, ce qui correspond à $f(|\vec{AP}|)\vec{AP}.\vec{G_A}$. Le vecteur gradient est $\vec{G_A} = \begin{pmatrix}(1\\0)\end{pmatrix}$.

FIGURE 17 – Fonction radiale et gradient de couleur

La fonction proposée par Perlin est $f(r) = (\frac{1}{2} - d)^4 * K$ où K est une constante qui compense les faibles valeurs obtenues suite à l'utilisation d'une puissance $4^{\text{ième}}$.

Il est relativement facile, connaissant les coordonnées des points du simplexe auquel appartient P de calculer la valeur du bruit en ce point. L'obtention de ces coordonnées n'est pas si évidente, comme nous allons le voir.

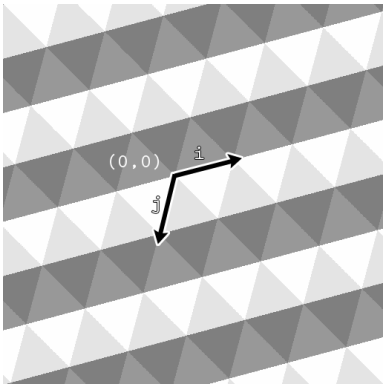
6.2 Changement de base

Les coordonnés des points que nous manipulons sont de la forme $\begin{pmatrix}(x\\y)\end{pmatrix}$ et sont écrits dans la base canonique (orthonormée directe) $(\vec{x}, \vec{y})$. Mais on peut tout à fait concevoir une base

5. De cette façon on s'assure, qu'en un point P, deux fonctions radiales ne peuvent s'additionner.

construite sur le pavage par des simplexes réguliers (en dimensions 2, ce sont des triangles équilateraux), de préférence composée de deux vecteurs unitaires [6]. La figure 18 présente la base dans laquelle nous souhaitons trouver les coordonnées du point P observé. En effet, si nous disposons de ses coordonnées dans cette base, il nous suffit de conserver la partie entière pour obtenir le sommet le plus proche de l'origine. Puisque chacun des vecteurs de notre base est une arrête, nous pouvons, avec ce point, en déduire tous les autres sommets constituant le simplexe.

FIGURE 18 – Pavage par des 2-simplexes réguliers de coté 1



Le changement de base *exact* fait intervenir des calculs correspondant à un produit matriciel. Dans le cas plan, les coefficients de la matrice sont des sinus et cosinus, et le calcul ne se simplifie pas élégamment. Nous allons donc utiliser une approximation de la véritable application "changement de base". Nous cherchons une application linéaire dont la matrice est de la forme

$$\begin{pmatrix} K+1 & \dots & K \\ \vdots & \ddots & \vdots \\ K & \dots & K+1 \end{pmatrix}$$

ceci afin que le calcul des nouvelles coordonnées se ramène au système

$$\begin{cases} x' &= x + K(x + y + \cdots + z) \\ y' &= y + K(x + y + \cdots + z) \\ \vdots & \\ z' &= x + K(x + y + \cdots + z) \end{cases}$$

où $x', y', \dots, z'$ sont les nouvelles coordonnées de P. L'avantage de ces equations est qu'il suffit de calculer un unique produit, pour obtenir par la suite chaque coordonné par une simple somme. De plus, une magnifique propriété de cette matrice est que son inverse (c'est-à-dire la matrice de l'application réciproque, qui permet de retourner de la nouvelle base à l'ancienne

6. Ils n'ont, bien entendu, aucune raison d'être orthogonaux.

base, afin de calculer les coordonnées de nos vecteurs dans la base canonique) est de la forme

$$\begin{pmatrix} 1-C & \dots & -C \\ \vdots & \ddots & \vdots \\ -C & \dots & 1-C \end{pmatrix}$$

où $C = \frac{K}{1+2K}$.

L'idée est que, si l'on pave notre premier repère, celui qui utilise la base canonique, avec des carrés, on peut alors constater l'existence de couples de triangles équilatéraux dans chacun d'eux. Alors chercher une application qui les transforme en triangles équilatéraux revient à chercher notre changement de coordonnées. Ken Perlin propose quelques valeurs pour K, présentées dans le tableau 19. Elles sont utilisées dans de nombreuses implémentations.

FIGURE 19 – Valeurs possible pour K

Dimension	Coefficient
2	$\frac{\sqrt{3}-1}{2}$
3	$\frac{1}{3}$
...	⋮
n	$\frac{\sqrt{n+1}-1}{n}$

6.3 Implémentation

Nous allons, une dernière fois, présenter une implémentation qui permettra certainement de poser les idées et peut-être d'éclaircir certains concepts. Nous nous appliquerons au cas bidimensionnel, qui constitue l'exemple le plus simple permettant d'illustrer le principe.

Listing 19 – Implémentation du simplex noise

```
////
//Fonction : Produit scalaire
////
double fast_dot(const int *v, const double x,
                const double y)
{
        return v[0] * x + v[1] * y;
}

////
//Fonction : Obtention du gradient pour un point P(x, y)
//             du plan
////
int *get_grad(int x, int y)
{
        int rand_value = perm[z + perm[y + perm[x]]];
```

```
        return _grad3[rand_value & 15];
}

////
//Fonction : Tronque la valeur x
//             et ne conserve que sa partie entière
inline int fastfloor(double x)
{
        return (x > 0) ? (int)x : (int)x - 1;
}

//La fonction principale
double smooth_noise_3d(double x, double y)
{
        //Définition des constantes K et C utilisées
        //dans les calculs de changement de base.
        const double K = (sqrt(3.) - 1.) / 2.;
        const double C = K / (1. + 2.*K);

        //On applique la transformation qui permet,
        // à partir des coordonnées (x,y),
        // d'obtenir les coordonnées (i,j)
        // dans la base des simplexes.
        double s = (x + y) * K;
        double i = x + s;
        double j = y + s;

        //On tronque les coordonnées de façon à obtenir le point
        // le plus proche de l'origine du simplexe contenant P
        i = fastfloor(i);
        j = fastfloor(j);

        //Nous effectuons le changement de base inverse,
        // c'est-à-dire qu'à partir des coordonnées (i,j)
        // d'un des sommets de notre simplexe, nous
        // cherchons les coordonnées (X0, Y0) dans
        // la base canonique.
        double t = (i + j) * C;
        double X0 = i - t;
        double Y0 = j - t;

        //Nous pouvons alors déterminer le premier vecteur
        // AP. Il reste à déterminer BP et CP.
        double x0 = x - X0;
        double y0 = y - Y0;

        //Nous devons déterminer si le point P se trouve
        // dans le triangle isocèle supérieur, ou bien
        // le triangle inférieur.
```

```
66          //Une fois çela déterminer, et en considérant
            // que le premier sommet est (0,0), nous savons
            // si le second sommet est (1,0) ou bien (0,1).
            // Nous stockons ses coordonnées dans (i1, j1)
            int i1 = 0, j1 = 0;
71          if(x0 > y0)
                    i1 = 1;
            else
                    j1 = 1;

76          //Nous pouvons alors déterminer les vecteurs BP et CP
            //En effet, si nous appliquons notre formule
            // aux vecteurs (0,1) et (1,0), nous
            // constatons que les coordonnées dans
            // la base caconique sont alors,
81          // respectivement, (-C, 1-C) et (1-C, -C).
            //Vecteur AP = (x1, y1)
            double x1 = x0 - i1 + C;
            double y1 = y0 - j1 + C;
            //Le troisième point est nécessairement le point (1,1)
86          // dont les coordonnées sont (1-2C, 1-2C).
            //Vecteur CP = (x2, y2)
            double x2 = x0 - 1 + 2.*C;
            double y2 = y0 - 1 + 2.*C;

91          //Nous calculons alors la norme de chacun de ces vecteurs :
            //|AP|
            double d0 = 0.5 - x0*x0 - y0*y0;
            //|BP|
            double d1 = 0.5 - x1*x1 - y1*y1;
96          //|CP|
            double d2 = 0.5 - x2*x2 - y2*y2;

            //Nous stockons le résultat du calcul dans la variable res
            double res = 0;
101
            //On applique un modulo 255 aux coordonnées i et j
            // afin de pouvoir déterminer leur gradient.
            int ii = (int)i & 255;
            int jj = (int)j & 255;
106
            //On ne calcule le produit scalaire que si les
            // fonctions radiales sont positives.
            //Dans ce cas, on effectue le produit scalaire,
            // exactement de la même façon qu'avec
111         // le bruit de Perlin.
            if(d0 > 0)
            {
                    d0 *= d0;
```

```
                res += d0 * d0 *
116                     dot(__get_grad(ii, jj), x0, y0);
        }
        if(d1 > 0)
        {
                d1 *= d1;
121             res += d1 * d1 *
                        dot(__get_grad(ii+i1, jj+j1), x1, y1);
        }
        if(d2 > 0)
        {
126             d2 *= d2;
                res += d2 * d2 *
                        dot(__get_grad(ii + 1, jj + 1), x2, y2);
        }

131     //On applique le facteur K permetant de ramener
        // l'amplitude de la valeur proche de [-1, 1].
        return 60 * res;
}
```

7 Conclusion

Vous disposez maintenant de toutes les connaissances nécessaires à l'implémentation d'un bruit de Perlin à N dimensions. Vous trouverez sur internet nombre de documents expliquant comment composer un bruit de Perlin avec d'autres fonctions afin d'obtenir des textures semblables à du bois, du marbre, du feu, et de nombreuses autres matières.

Si vous souhaitez en apprendre plus sur les alternatives au bruit de Perlin, je vous recommande chaudement de vous renseigner au sujet du *bruit de Gabor* (cf:[1]).

7.1 Remerciements

Je tiens à remercier Mael Minot pour sa relecture orthographique.

Références

[1] G. Drettakis P. Dutré A. Lagae, S. Lefebvre. Procedural noise using sparse gabor convolution. http://graphics.cs.kuleuven.be/publications/LLDD09PNSGC/LLDD09PNSGC_paper.pdf.

[2] Paul Bourke. Interpolation methods. Decembre 1999. http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/interpolation/.

[3] Paul Bourke. Perlin noise and turbulence. Janvier 2000. http://local.wasp.uwa.edu.au/~pbourke/texture_colour/perlin/.

[4] Hugo Elias. Perlin noise. Decembre 1998. http://freespace.virgin.net/hugo.elias/models/m_perlin.htm.

[5] Wikipedia EN. Interpolation bicubique. http://en.wikipedia.org/wiki/Bicubic_interpolation.

[6] Wikipedia EN. Interpolation tricubique. http://en.wikipedia.org/wiki/Tricubic_interpolation.

[7] Wikipedia EN. Perlin noise. http://en.wikipedia.org/wiki/Perlin_noise.

[8] François Faure. Interpolation de positions-clefs. http://www-evasion.imag.fr/~Francois.Faure/enseignement/maitrise/interpolation/interpolation.pdf.

[9] Stefan Gustavson. Simplex noise demystified. Mars 2005. http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

[10] Ken Perlin. Improving noise. http://mrl.nyu.edu/~perlin/paper445.pdf.

[11] Ken Perlin. Making noise. December 1999. http://www.noisemachine.com/talk1/index.html.

[12] Iñigo Quilez. Advanced perlin noise. 2008. http://iquilezles.org/www/articles/morenoise/morenoise.htm.

[13] Pierre Schwartz. Génération de terrain par l'algorithme de perlin. http://khayyam.developpez.com/articles/algo/perlin/.

Index