Rapport ERDI

Cyril Colin
Voisin Dylan

Master 1 Informatique
Université d'Aix-Marseille

Décembre 2019

Table des matières

Introduction

Dans le monde de l'Infographie et plus particulièrement dans le domaine de la modélisation, les textures sont devenues indispensables pour associer un modèle et une image, afin de créer un objet réaliste, avec sa forme et ses motifs, apportés par les textures. Les domaines d'application sont nombreux, l'animation, les images de synthèses utilisées dans les films, ou encore les jeux vidéos. Plus les textures sont précises (et donc par conséquent lourde en terme d'espace disque), plus la qualité du rendu final sera réaliste.

Il se pose par conséquent une question de limitations, en effet, ces textures de très hautes qualités prennent certes de la place, mais également du temps à concevoir et à créer. Un autre problème peut également intervenir, même si l'on veut une texture très belle et que l'on peut y arriver, on aimerait dans l'idéal avoir plusieurs textures pour un type d'objet donné. Prenons l'exemple des jeux vidéos, il serait assez facheux de n'avoir que 3 ou 4 textures pour les arbres, donnant ainsi une diversité assez pauvre et une répétition qui peut vite lasser l'utilisateur, qu'importe la qualité de rendu de ces dites textures.

Avoir suffisamment de textures debonne qualité prendrait beaucoup trop de temps et de place pour pouvoir être créé. Il faudrait donc palier à cela en utilisant d'autres procédés. Fort heureusement, ces procédés existent et se basent pour la plupart sur des particularités présentes dans la nature, et un point commun entre les objets visibles dans la nature est bien entendu le hasard. De plus, les objets et formes de même type ont des propriétés communes entre elles. De ce fait, nous sommes en mesure de créer des fonctions mathématiques basée sur l'aléa capables de s'adapter, moyennant certains paramètres, à plusieurs textures présentes dans la nature. Pour rendre le tout encore plus attractif, ceci se fait de manière procédurale.

Avec cette méthode, nous sommes donc capables de palier non seulement à la qualité mais aussi à la diversité des textures concevables, le tout de manière presque instantannée et sans prendre d'espace disque. Ce type de texture est plus communément appelé « textures procédurales » et permet la synthèse de textures comme le bois, le marbre, le granite, le métal...

Ces textures procédurales sont cependant générées avec des procédés différents selon le type de texture que l'on souhaite obtenir, ces procédés

sont appelés les générateur de bruits, puisque basés sur l'aléa. On notera cependant que certaines méthodes telle que le texturage cellulaire ne se base pas sur un générateur de bruit.

Dans ce rapport, nous allons donc voir différents types de générateurs de bruits, on se limitera cependant aux générateurs de bruits gradients, par oppositions aux bruits de valeurs. Dans un premier temps, nous verrons l'exemple du bruit de Perlin, s'en suivra le bruit de Gabor qui a inspiré notre dernière partie qui se concentre autour du sujet d'étude : le « Phasor Noise ».

1 Bruit de Perlin

Dans cette partie, nous nous intéresseront au bruit de Perlin. Pour le contexte de la découverte de ce générateur de bruit, voici un extrait de l'article Wikipédia dédié :

« Le bruit de Perlin a été développé par Ken Perlin en 1985. À cette époque, après avoir travaillé sur les effets spéciaux de Tron pour MAGI en 1981, il cherchait à éviter le look « machinique ». Il commença donc par mettre au point une fonction pseudo-aléatoire de bruit qui remplit les trois dimensions de l'espace, avant d'inventer l'année suivante le premier langage de shading. Ses travaux sur les textures procédurales ont valu à Ken Perlin l'Academy Award for Technical Achievement en 1997. »

Une des particularité de sa méthode est que les détails du bruit généré sont de même taille, ce qui permet de combiner une texture générée d'une taille donnée avec d'autres échelles pour ajouter de la profondeur.

1.1 Principe

L'algorithme du bruit de Perlin se décompose en 3 parties que sont :

1. la définition de la grille.
2. le calcul du produit scalaire entre le vecteur gradiant et le vecteur distance.
3. l'interpolation entre ces valeurs.

Pour la définitionde la grille, il faut définir une grille à n dimensions. Attribuer pour chaque nœud un vecteur de gradient aléatoire de norme 1 et de dimension n.

Pour ce qui concerne le produit scalaire, nous faisons comme suit :

Soit un point de l'espace à n-dimensions envoyé à la fonction de bruit, l'étape consiste à déterminer dans quelle cellule de grille le point donné se situe. Pour chaque nœud-sommet de cette cellule, calculer le vecteur distance entre le point et le nœud-sommet. Puis calculer le produit scalaire entre le vecteur de gradient au nœud et le vecteur de distance. Cela conduit à l'échelle de complexité $O(2^n)$.

La dernière étape est l'interpolation entre les 2^n produits scalaires calculés aux nœuds de la cellule contenant le point d'argument. Cela a pour conséquence que la fonction de bruit renvoie 0 lorsqu'elle est évaluée sur les nœuds de la grille eux-mêmes.

L'interpolation est effectuée en utilisant une fonction dont la dérivée première (et éventuellement la dérivée seconde) est nulle aux 2^n nœuds de la grille. Cela a pour effet que le gradient de la fonction de bruit résultante à chaque nœud de grille coïncide avec le vecteur de gradient aléatoire précalculé.

Voici le pseudo-code de l'implantation du bruit de Perlin à 2 dimensions :

```
// Function to linearly interpolate between a0 and a1
// Weight w should be in the range [0.0, 1.0]
function lerp(float a0, float a1, float w) {
    return (1.0 - w)*a0 + w*a1;
}

// Computes the dot product of the distance and gradient vectors.
function dotGridGradient(int ix, int iy, float x, float y) {

    // Precomputed (or otherwise) gradient vectors at each grid r
    extern float Gradient[IYMAX][IXMAX][2];

    // Compute the distance vector
    float dx = x - (float)ix;
    float dy = y - (float)iy;

    // Compute the dot-product
    return (dx*Gradient[iy][ix][0] + dy*Gradient[iy][ix][1]);
}

// Compute Perlin noise at coordinates x, y
```

```
function perlin(float x, float y) {

    // Determine grid cell coordinates
    int x0 = floor(x);
    int x1 = x0 + 1;
    int y0 = floor(y);
    int y1 = y0 + 1;

    // Determine interpolation weights
    // Could also use higher order polynomial/s-curve here
    float sx = x - (float)x0;
    float sy = y - (float)y0;

    // Interpolate between grid point gradients
    float n0, n1, ix0, ix1, value;
    n0 = dotGridGradient(x0, y0, x, y);
    n1 = dotGridGradient(x1, y0, x, y);
    ix0 = lerp(n0, n1, sx);
    n0 = dotGridGradient(x0, y1, x, y);
    n1 = dotGridGradient(x1, y1, x, y);
    ix1 = lerp(n0, n1, sx);
    value = lerp(ix0, ix1, sy);

    return value;
}
```

Listing 1 – Implantation du bruit de Perlin en pseudo-code