

Algorithmique avancée

Licence 3 – 2016

Projet :

« Limiter le nombre de couleurs par une adaptation de l'algorithme de Voronoi. »

Cardinal Dorian

Analyse du sujet

Le projet consiste à compresser une image, et à pouvoir la décompresser ensuite. La méthode de compression est la color quantization, et l'algorithme utilisé est une adaptation de l'algorithme de Voronoi.

Il existe plusieurs algorithmes pour effectuer un diagramme de voronoi, le plus naïf étant la recherche du spot le plus proche pour chaque pixels, puis le remplacement de chaque pixel par la couleur du spot associé.

Il est possible d'appliquer directement cet algorithme à une image, en choisissant la couleur de chaque spot en fonction des pixels associés à celui-ci.

Cette méthode permet effectivement de réduire le nombre de couleurs de l'image, cependant les formes ne seront pas conservés, puisque déterminés aléatoirement en fonction des spots.

Pour retrouver du détail, il faut des zones plus petites, donc augmenter le nombre de spots. Cela augmente énormément le temps de calcul (puisque'on parcours l'ensemble des spots pour chaque pixel) et le niveau de compression ne serait pas très haut.

En appliquant cet algorithme à l'ensemble des couleurs plutôt que sur l'ensemble des pixels, il est possible de réduire le nombre de couleurs. L'idée est la suivante : On dispose de 3 axes, un pour chaque composant de la couleur (rouge, vert, bleu). Les pixels de l'image ainsi que les spots peuvent y être représentés en fonction de leur couleur, en ignorant leur position dans l'image.

Première implémentation

On cherche dans un premier temps à modifier l'image

```
#include "ima.h"
#define NB_POINTS 15

int site[NB_POINTS][3];
int allcolors[256][256][256];

int nearest_site(double x, double y, double z)
{
    int i, ret = 0;
    double d, min = 0;

    for(i = 0; i < NB_POINTS; i++)
    {
        d = (x - site[i][0]) * (x - site[i][0]) + (y - site[i][1]) * (y - site[i][1]) + (z - site[i][2]) * (z - site[i][2]);
        if(!i || d < min)
            min = d, ret = i;
    }
    return ret;
}

void voronoi(Image *img)
{
    for(int i = 0; i < NB_POINTS; i++)
    {
        GLubyte *pos = img->data + (((rand() % img->sizeY) * img->sizeX) + (rand() % img->sizeX)) * 3;
        site[i][0] = pos[0];
        site[i][1] = pos[1];
        site[i][2] = pos[2];
    }

    for(int z = 0; z < 256; z++)
    {
        for(int y = 0; y < 256; y++)
        {
            for(int x = 0; x < 256; x++)
                allcolors[z][y][x] = nearest_site(x, y, z);
        }
    }

    GLubyte *im = img->data;
    for(int i = 0; i < img->sizeX * img->sizeY; i++)
    {
        im[0] = site[ allcolors[im[0]][im[1]][im[2]] ][0];
        im[1] = site[ allcolors[im[0]][im[1]][im[2]] ][1];
        im[2] = site[ allcolors[im[0]][im[1]][im[2]] ][2];
        im += 3;
    }
}
```

On commence par générer nos sites aléatoirement (il s'agit de selectionner NB_POINTS triplets RGB qui existent sur l'image).

Puis, on construit l'ensemble des différentes couleurs possible avec un octet par couleur (true color).

La couleur sera identifiée par son index : allcolors[r][g][b].

On attribut à chaque couleur le numéro du site le plus proche.

Enfin, on parcourt tous les pixels de l'image, on récupère le site correspondant et on modifie la couleur du pixel par la couleur du site.

Cet algorithme est très lent pour deux raisons : On visite l'ensemble des couleurs possibles, et on accède à la mémoire via un tableau multidimensionnel, qui est une opération lente en assembleur.

Compression et décompression

La première amélioration consiste donc à convertir nos 3 Glubyte en entiers (de 0 à 0xFFFF), et créer un tableau permettant de stocker les index de nos spots.

```
int *allcolors = malloc(sizeof(int) * 0x1000000);
```

Les éléments sont de type entier car il peut y avoir plus de 256 spots différents.

Ensuite, on va essayer de déterminer si la couleur est utilisée ou non, afin de calculer le spot correspondant uniquement dans le cas où nous visiterions la couleur pour la première fois. On admettra la valeur -1 comme étant l'absence de spot. Ou pourrait aussi utiliser un autre tableau de char qui tiendrait dans 16MO, ou bien implémenter un bitarray si les temps d'exécution n'étaient pas une des priorités.

La première partie de l'algorithme consiste à parcourir l'ensemble des pixels de l'image, obtenir un entier permettant d'accéder aux tableaux allcolors, et si un site n'a jamais été calculé pour cette couleur, lui en attribuer un, écrire dans une chaîne de caractère le spot.

```
out = malloc(sizeof(unsigned char) * size_img);
for(i = 0; i < size_img; i++)
{
    couleur = (im_ptr[0] << 16) + (im_ptr[1] << 8) + im_ptr[2];
    if(allcolors[couleur] == -1)
    {
        site = nearest_site(sites, nb_sites, im_ptr[0], im_ptr[1],
im_ptr[2]);
        allcolors[couleur] = site;
    }
    out[i] = allcolors[couleur];
    im_ptr += 3;
}
```

La deuxième partie de l'algorithme consiste à faire un RLE (https://fr.wikipedia.org/wiki/Run-length_encoding) sur le fichier de sortie avec la chaîne out.

```
for(i = 1, j = 1, prec = out[0]; i < size_img; i++)
{
    if(out[i] != prec || j == 0xFF)
    {
        fwrite(&j, 1, 1, f_out);
        fwrite(&prec, size_spots, 1, f_out);
        j = 0;
    }
    j++;
    prec = out[i];
}
fwrite(&j, 1, 1, f_out);
fwrite(&prec, 1, size_spots, f_out);
```

Le nombre de pixels successifs de même couleur s'écrira sur un seul octet, tandis que l'index du spot peut s'écrire sur 1 octet (si le nombre de spots est inférieur à 255) ou sur deux octets (avec un nombre limité de spots de 0x1000).

```
size_spots = 1 + (nb_sites > 0xFF);
```

On peut accélérer légèrement le temps d'exécution et se libérer de la chaîne out en fusionnant les deux parties ainsi :

```
couleur = (im_ptr[0] << 16) + (im_ptr[1] << 8) + im_ptr[2];
prec = allcolors[couleur] = site = nearest_site(sites, nb_sites, im_ptr[0],
im_ptr[1], im_ptr[2]);
im_ptr += 3;
for(i = 1, j = 1; i < size_img; i++, j++, im_ptr += 3, prec = site)
{
    couleur = (im_ptr[0] << 16) + (im_ptr[1] << 8) + im_ptr[2];
    if(allcolors[couleur] == -1)
        allcolors[couleur] = nearest_site(sites, nb_sites, im_ptr[0],
im_ptr[1], im_ptr[2]);

    site = allcolors[couleur];
    if(site != prec || j == 0xFF)
    {
        fwrite(&j, 1, 1, f_out);
        fwrite(&prec, size_spots, 1, f_out);
        j = 0;
    }
}
fwrite(&j, 1, 1, f_out);
fwrite(&prec, size_spots, 1, f_out);
```

Pour pouvoir interpréter ces couplets (nombre de pixels, numéro du spot) et écrire une image à partir de ces données, on va avoir besoin des caractéristiques suivantes : largeur et hauteur de l'image d'origine, nombre de spots et couleur de chacun des spots. On rajoutera 2 caractères en en-tête permettant d'identifier le format de l'image.

```
char *format = "VC";
fwrite(format, 1, 2, f_out);
fwrite(&img->sizeX, 8, 1, f_out);
fwrite(&img->sizeY, 8, 1, f_out);
fwrite(&nb_sites, 2, 1, f_out);
for(i = 0; i < nb_sites; i++)
    fwrite(sites[i], 1, 3, f_out);
```

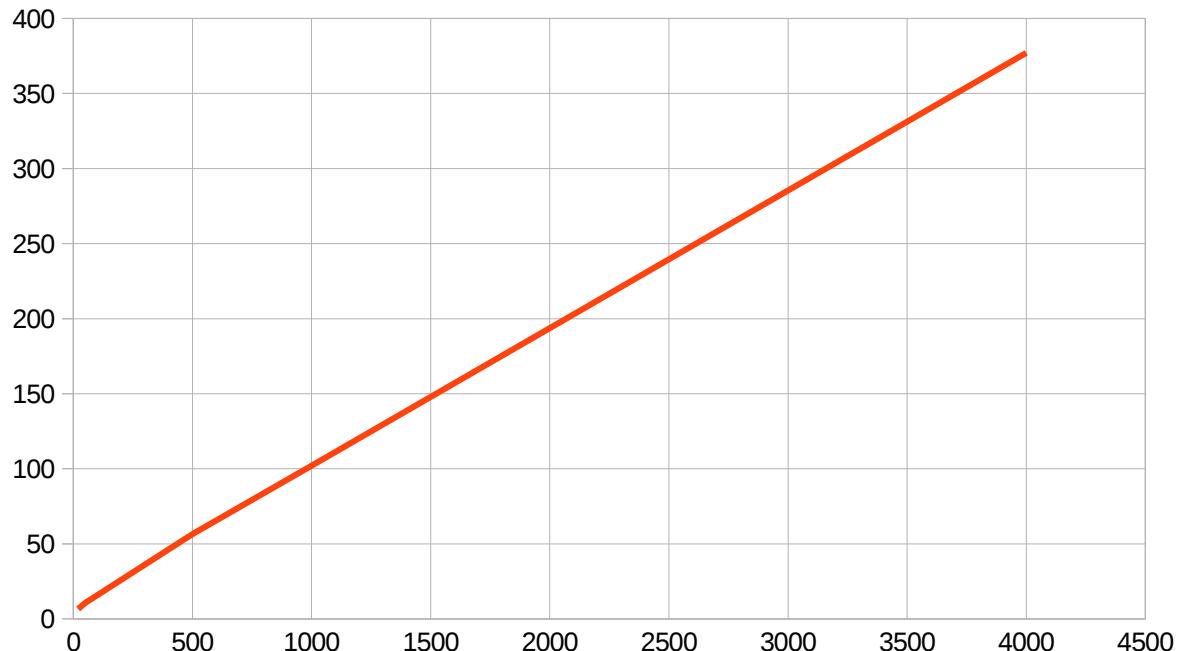
La décompression se fera donc ainsi : On vérifie le format > On récupère la taille de l'image et le nombre de spots > On alloue un tableau pour contenir les spots > On lit la couleurs des spots qu'on stock dans le tableau > On remplit les pixels de l'image avec les couplets > On retourne l'image et on la sauvegarde.

```
while(fread(&nb_pixel, 1, 1, f_in) && fread(&site, size_spots, 1, f_in))
{
    while(nb_pixel--)
    {
        img_ptr[0] = sites[site][0];
        img_ptr[1] = sites[site][1];
        img_ptr[2] = sites[site][2];
        img_ptr += 3;
    }
}

for(site = 0; site < nb_sites; site++)
{
    sites[site] = malloc(sizeof(GLubyte) * 3);
    fread(sites[site], 1, 3, f_in)
}
```

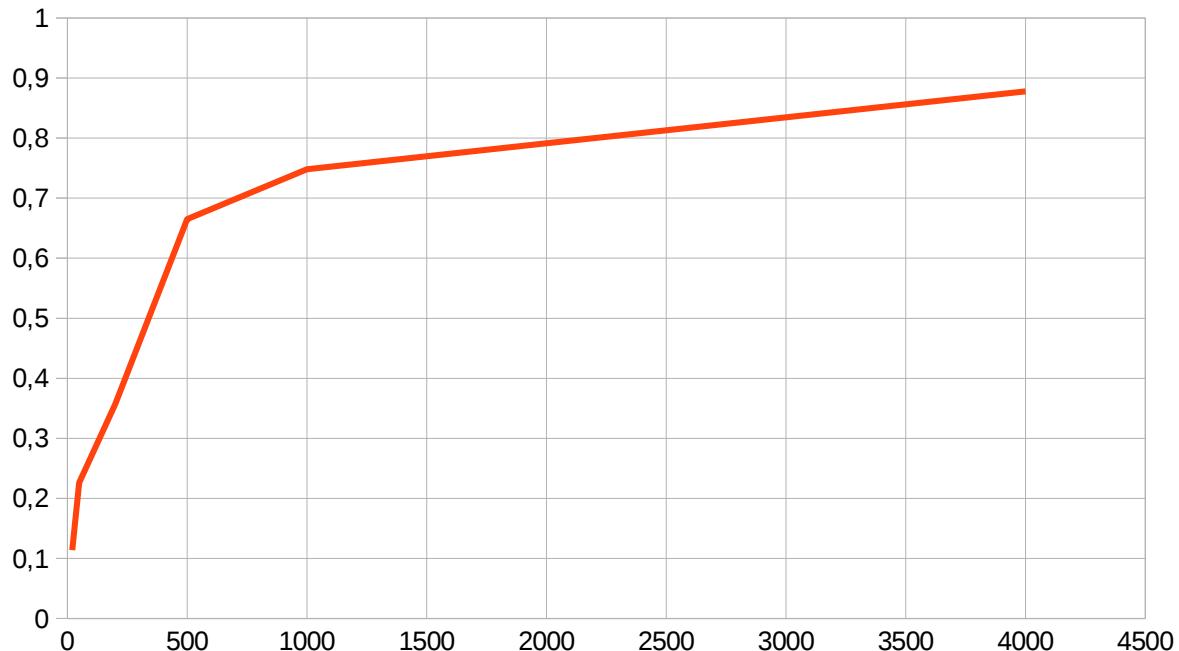
Évaluation de l'algorithme

Compression : Evolution du temps d'exécution en fonction du nombre de spots (x), pour 100 exécutions de l'algorithme :



L'évolution est constante car la taille de l'image ne varie pas, et que pour chaque couleur unique de l'image nous parcourons en moyenne $\text{nb_spots}/2$ spots.

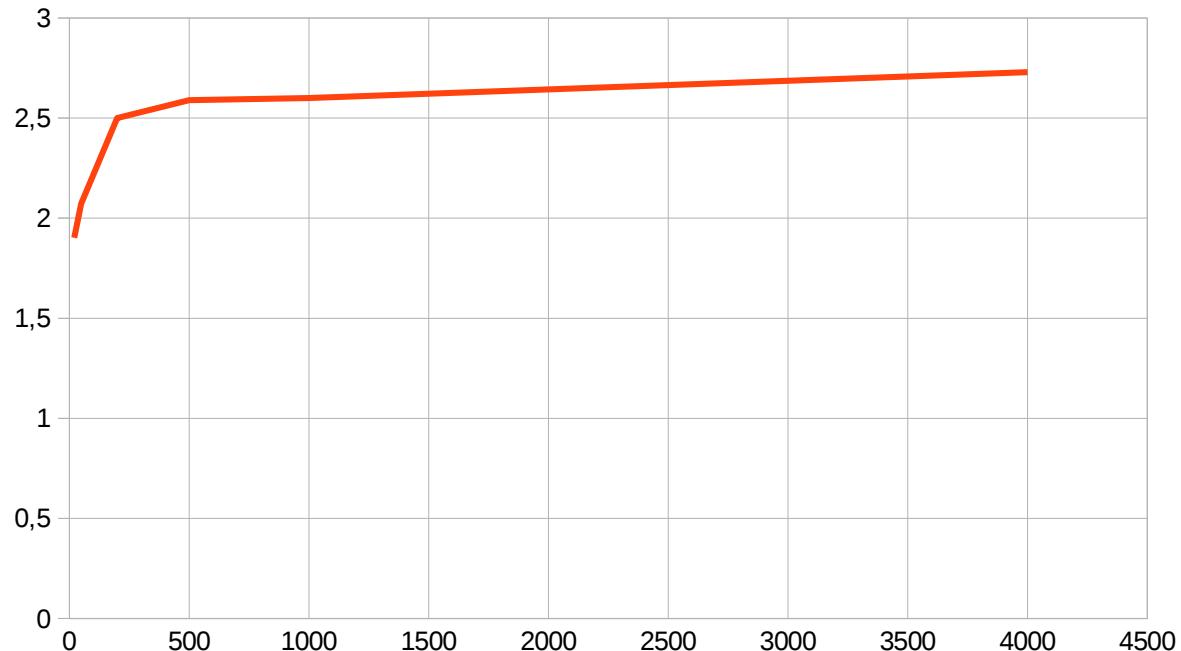
Image compressée : Evolution du ratio de compression en fonction du nombre de spots :



En dessous de 256 spots, l'index du spot s'écrit sur un seul octet. De plus, le faible nombre de couleurs fait que les pixels se suivent, on peut donc résumer un grand nombre de pixels avec 2 octets.

Quand le nombre de spots se rapproche du nombre de couleurs de l'image, chaque couplet (2 octets pour l'index du spot, 1 octet pour le nombre de pixels identiques successifs), la compression devient inutile (le ratio de compression tend vers 1) car écrire un couplet revient à décrire un pixel.

Décompression : Evolution du temps d'exécution en fonction du nombre de spots, pour 100 exécutions de l'algorithme :



L'influence du nombre de spots sur le temps de décompression est presque nul. Cela représente une lecture de $3 * \text{nb_spots}$ octets. En fait, la quasi totalité du temps d'exécution correspond au temps d'écriture de l'image décompressée sur le disque, qui va dépendre de la vitesse d'écriture et de la taille de l'image. En réalité, dans beaucoup de cas, l'image décompressée n'est même pas écrite sur le disque mais uniquement stockée dans la mémoire vive (les textures compressées dans les jeux vidéos par exemple).

Qualité de l'image :

Image de base :



Image compressée avec 20 couleurs, seed 0 :



Du fait que les spots soient sélectionnés aléatoirement, les chances de sélection d'une couleur vont dépendre de sa fréquence d'apparition dans l'image et du nombre de spots.

Avec peu de spots, une petite tache de couleur unique aura beaucoup de chances de changer de couleur.

Aucun pixel rouge n'a été sélectionné, tous les rouges sont transformés en la couleur la plus proche sélectionnée, ici le gris.

Même compression, en utilisant le seed 7 :



Par chance, un rouge a été sélectionné.

Image de base :

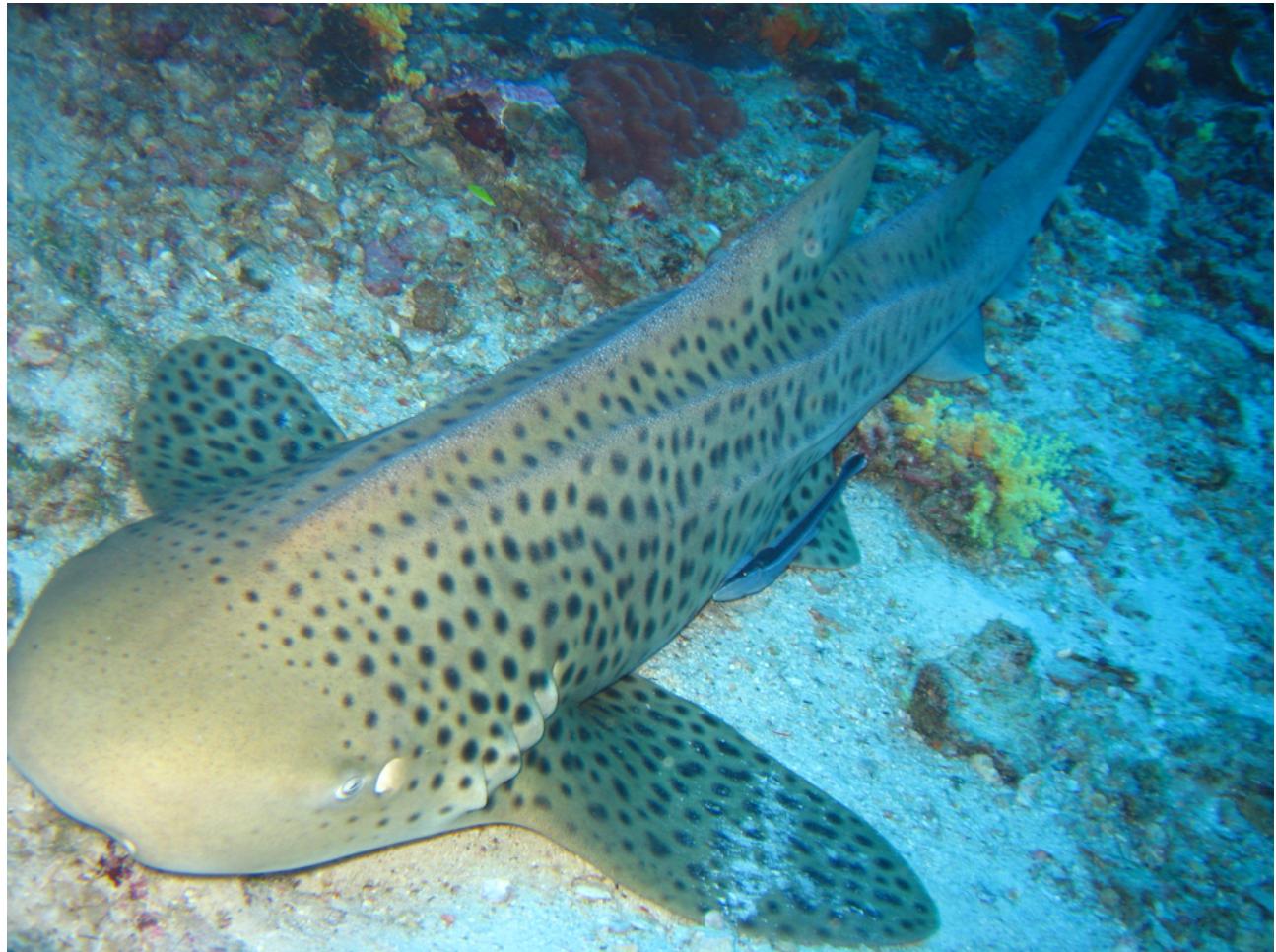
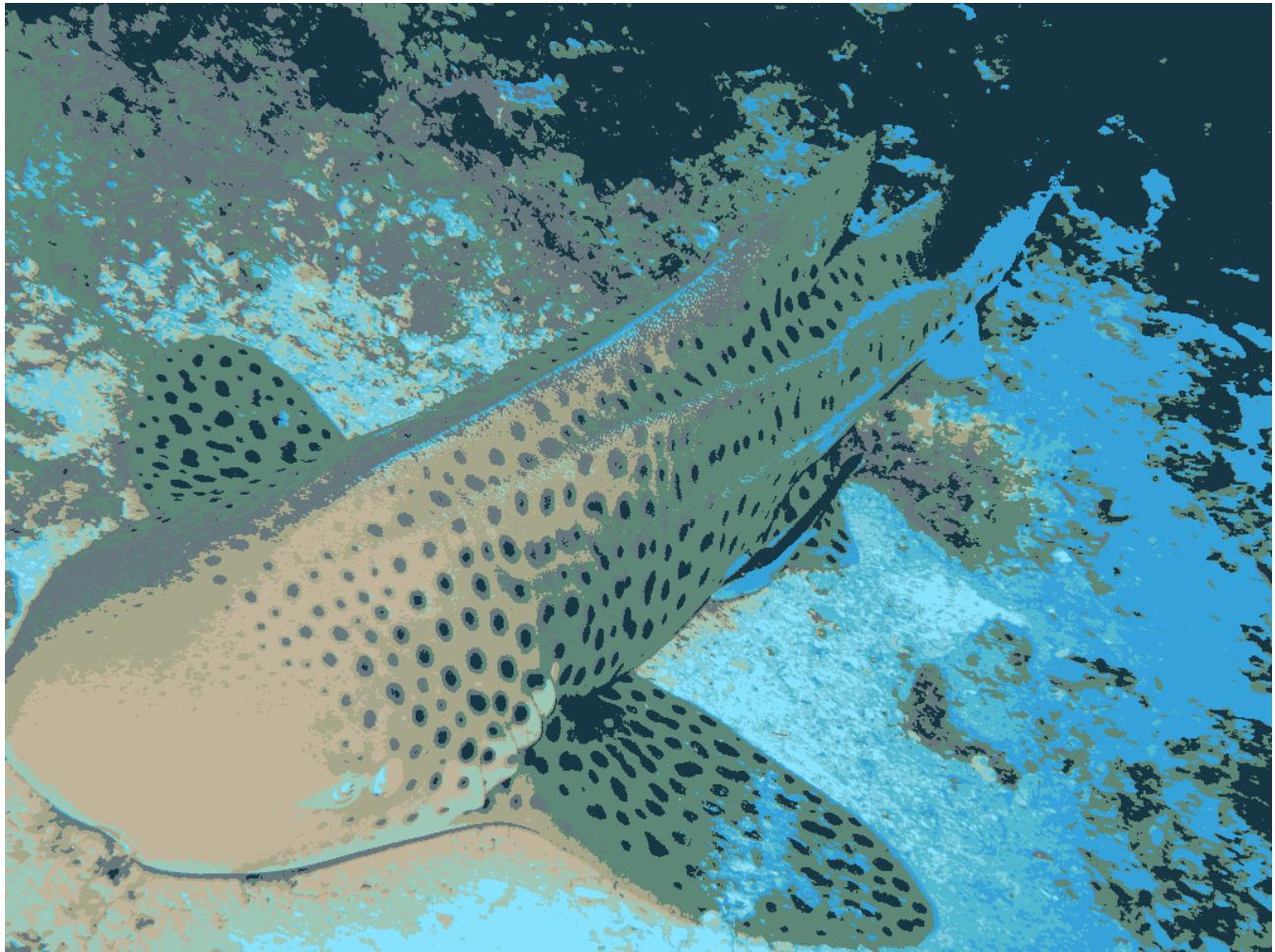
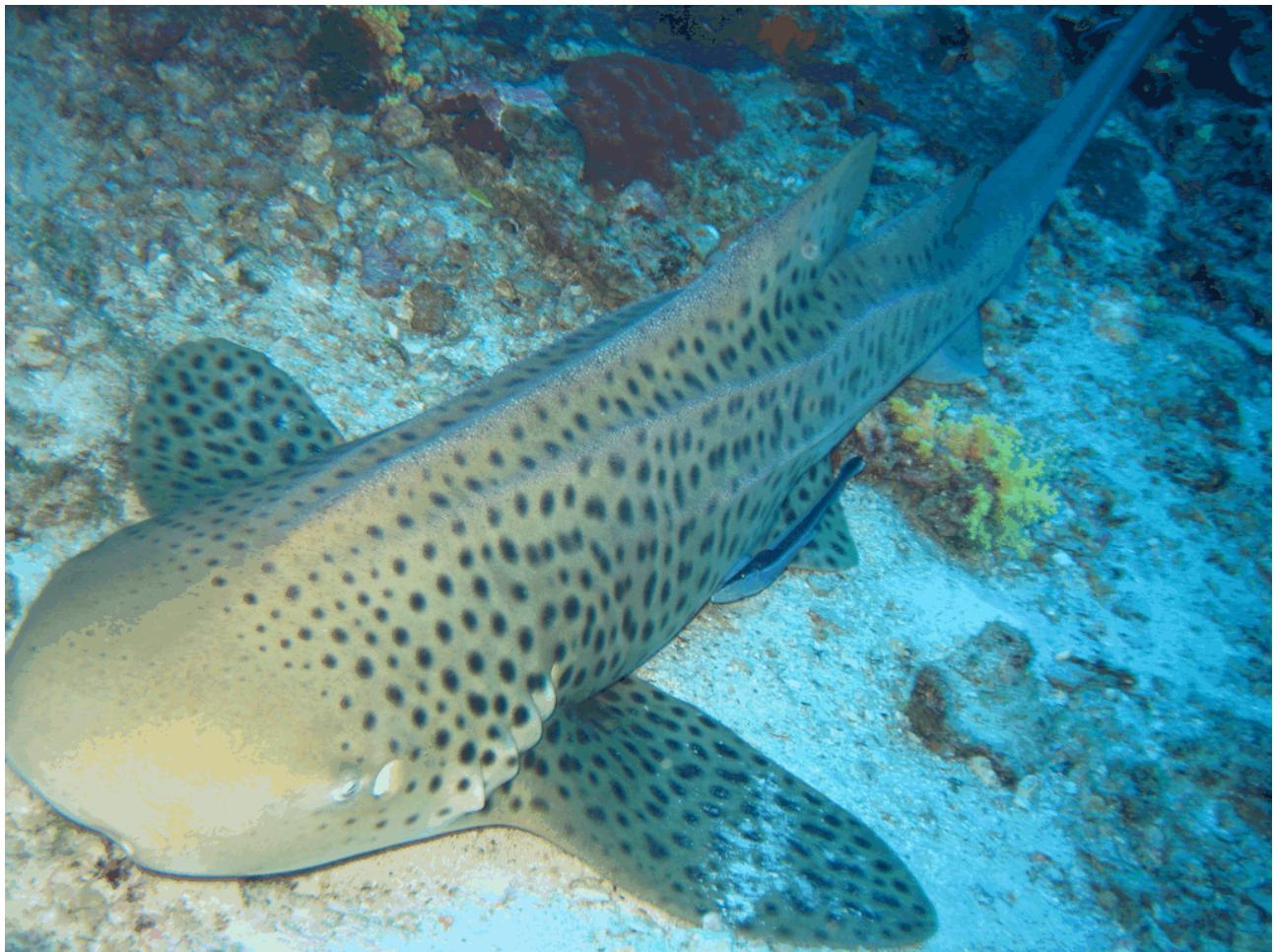


Image compressée avec 10 couleurs, seed 254 :



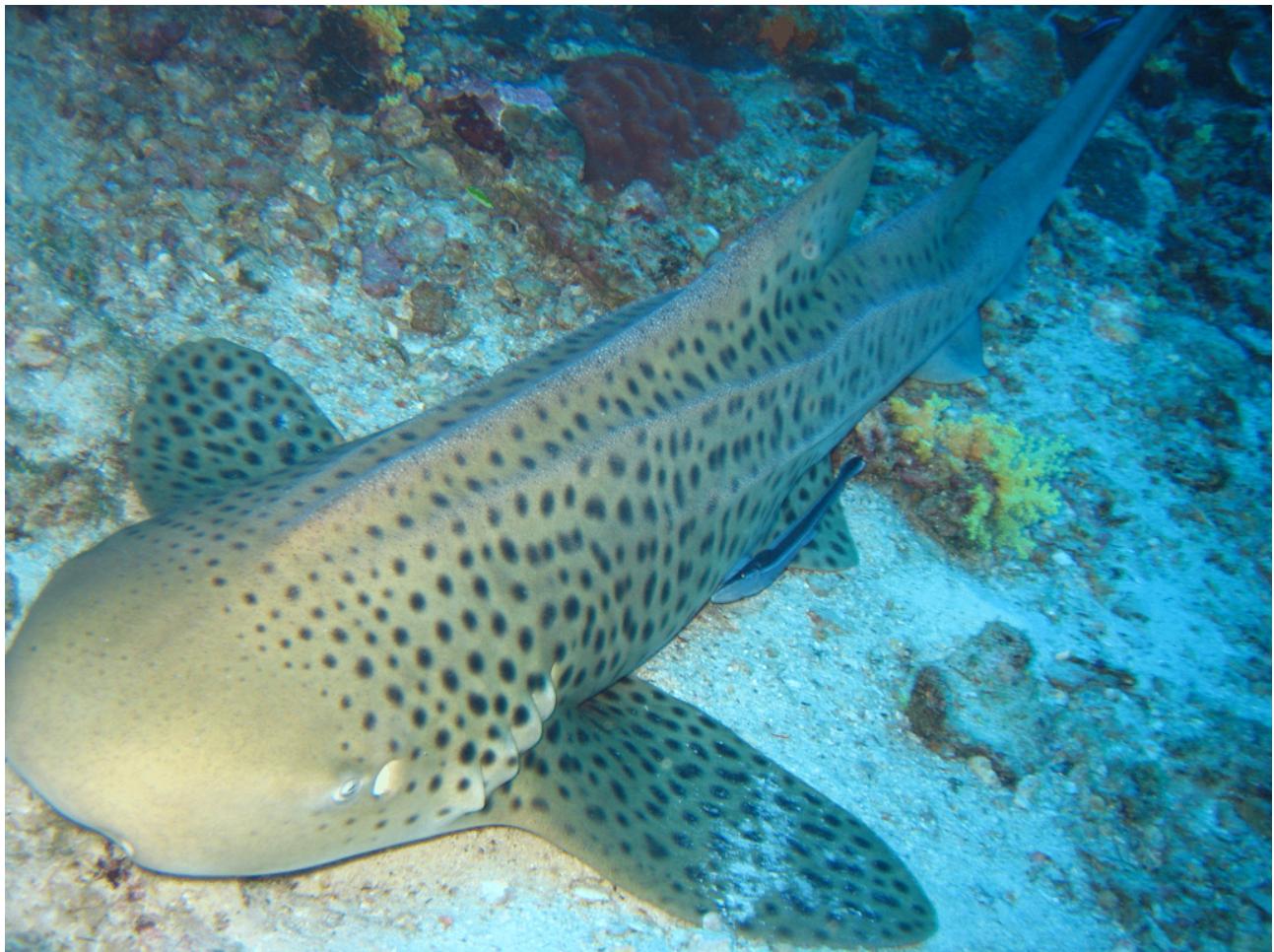
On perd beaucoup de détails, mais les tâches sur le poisson sont très jolies.

Avec 255 couleurs :



On retrouve du détail, à part dans les dégradés. L'image compressée pèse 37 % du poids de l'image de base.

Avec 4000 couleurs :



Les différences avec l'image de base sont à peine perceptibles, mais la compression est très faible (10% environ)

Améliorations possibles

- Le principal problème vient de la sélection des spots, l'aléatoire n'est pas très efficace. Une amélioration évidente consiste donc à utiliser un algorithme permettant d'extraire une palette de nb_spots couleurs de l'image. Median cut semble approprié dans ce cas.
- Avec un grand nombre de spots (à partir de 1000 environ), on peut directement noter l'index du spot et ignorer la compression RLE. Le fichier généré pèserait alors invariablement 75 % de l'image d'origine.
- Il est possible d'utiliser une autre méthode pour calculer la distance entre deux couleurs, plus proche de la perception humaine.
- Permettre la lecture et l'écriture d'autres formats d'images.