

Compression d'image par BSP Algorithmique Avancée

UP8: Semestre 6

Etienne PENAULT

Sommaire

1. Introduction	3
2. Utilisation	3
2.1. Compilation	3
2.2. Exécution	4
3. Fonctionnement	5
3.1. Structures Utilisées	5
3.1.a. Image	5
3.1.b. Nuage de Couleurs	5
3.1.c. Représentation dans l'Espace	6
3.1.d. Arbre Binaire de Partitionnement de l'Espace	7
3.1.e. Table de Correspondance des Couleurs	8
3.2. Partition Binaire de l'Espace	8
3.3. Principe de la C-LUT	14
3.4. Sélection des Nouvelles Couleurs	15
3.5. Création de l'Image Compressée	17
3.6. Ecriture et Lecture de l'Image Compressée	19
3.6.a. Ecriture	19
3.6.b. Lecture	22
4. Analyse	23
4.1. Taux de Compressions	23
4.2. Perte de Qualité	24
4.3. Temps d'Exécution	27
5. Difficultés	28
6. Conclusion	28
7. Remerciements	29
8. Sources	29

1. Introduction

La partition binaire de l'espace, de son nom anglais Binary Space Partitioning (BSP) est, comme son nom l'indique, un principe de sectionnement de l'espace. En effet, BSP repose sur le principe de divisions successives d'un espace en le séparant en deux, créant alors des sous espaces qui seront à leurs tour sectionnés en deux autres plus petits, et ainsi de suite. L'utilisation de BSP est très polyvalente par son principe assez basique. Il peut être utilisé dans le domaine des jeux pour générer des pièces avec des salles, comme il peut être utilisé dans des choses plus poussées tel que des calculs de collisions ou encore d'ombres. D'ailleurs, BSP fût utilisé pendant un moment dans les anciennes pipelines graphiques, mais ces dernières sont maintenant obsolètes.

Dans le cadre de ce projet, nous allons utiliser BSP sur l'espace des couleurs de l'image afin de réaliser une compression. Cela reviendra donc à utiliser un nombre de couleurs restreint afin de "simplifier" notre palette de couleurs à utiliser pour la nouvelle image compressée.

Ensuite nous interpréterons les résultats obtenus afin d'en tirer des conclusions.

2. Utilisation

2.1. Compilation

Le compilateur utilisé est GCC et la compilation est crossplatform Linux/MacOS. Pour compiler le programme, 2 options de types sont proposées:

- GLubyte
- Unsigned Short

Effectivement, nous pouvons choisir le type à utiliser dans notre programme car l'image compressée dépendait du type utilisé à la compilation dans des versions antérieures du projet. Cependant, l'option est laissée volontairement, car si nous voulons utiliser seulement 256 couleurs, des GLubyte sont amplement suffisants pour stocker les couleurs, c'est une petite optimisation.

La compilation sera par défaut faites avec des Unsigned Short.

Compilation avec GLubyte:

```
1 make TYPE=-DGLUBYTE
```

Compilation avec Unsigned Short:

```
1 make TYPE=-DSHORT
```

2.2. Exécution

Ici, la commande générique pour exécuter le programme:

```
1 ./exécutable image nombre_de_passe_de_divisions_des_espaces
→ axe_de_la_coupe_dans_notre_espace_de_couleur placement_de_la_coupe_en_pourcentage
```

Les arguments tels qu'ils apparaissent ne sont pas très explicites, mais ils seront détaillés au court du rapport (notons que les axes X, Y, et Z correspondent à 0, 1, 2). En voici un exemple:

```
1 ./build/bin/bsp.out ./img/Refuges.ppm 8 0 50
```



Figure 1: Image originale

possède une option de prévisualisation de la future image compressée, afin de se rendre compte de la perte potentielle de qualité avant de vouloir compresser la nouvelle image.

Sur la Figure 2, nous pouvons voir un aperçu de notre image post-compression. Avant de rentrer dans le principe de la compression, regardons brièvement à quoi ressemble le résultat. Nous pouvons tout à fait reconnaître l'image originale. La subtilité de la compression dites par "quantization" (dont fait partie la compression par BSP) est que nous allons reduire le nombre de couleurs utilisées dans l'image. J'attire votre attention sur ce qui a été expliqué dans la partie 2.2.. Nous avons choisi 8 divisions successives, nous aurrons donc $2^8 = 256$ couleurs utilisées dans notre nouvelle image.

L'image originale s'affiche. On pourra par la suite sélectionner les options qui nous intéressent par un clic gauche sur la fenêtre. Ensuite, nous pourrons sélectionner soit de compresser l'image affichée à l'écran, soit de décompresser une image compressée au préalable par ce même programme. La saisie du nom de l'image à compresser ou décompresser à lieu au niveau du terminal et non au niveau de la fenêtre. Lors de la compression, un nouveau fichier sera créé à la racine du projet, avec les informations nécessaires pour décompresser ce même fichier (comme les couleurs, la taille... Mais nous y reviendrons plus tard). De plus, le programme

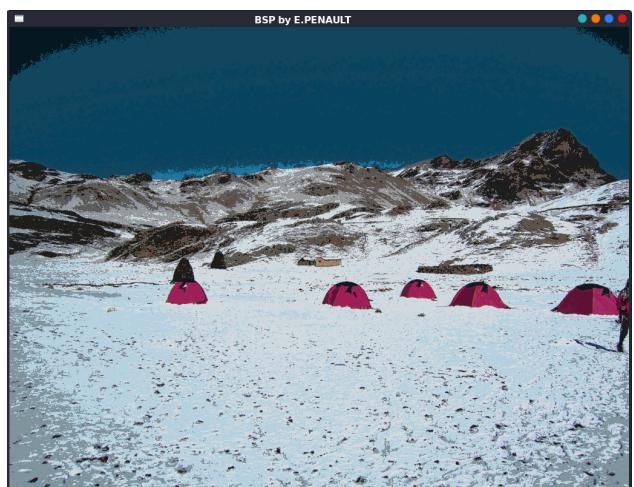


Figure 2: Prévisualisation de la compression

3. Fonctionnement

3.1. Structures Utilisées

Avant de rentrer dans les détails du fonctionnement du code, faisons tout d'abord un point sur les structures utilisées dans le programme.

3.1.a. Image

```

1 struct Image {
2     unsigned long sizeX;
3     unsigned long sizeY;
4     GLubyte *data;
5 };
6 typedef struct Image Image;
7
8 struct compressedImage {
9     unsigned long sizeX;
10    unsigned long sizeY;
11    usedType *data;
12 };
13 typedef struct compressedImage CompressedImage;

```

Nous possédons deux structures pour représenter les images dans notre code.

La première stocke les dimensions de l'image, ainsi que les pixels dans un tableau de GLubyte. Nous stockons les pixels dans ce tableau, composante par composante. C'est à dire que nous avons un pas de trois entre chaque pixel car nous stockons à la suite la composante Rouge, Verte et Bleu de chaque pixel.

La seconde structure est semblable à la première à une exception près. Notre tableau stockant les données de l'image n'est pas de type GLubyte mais de type usedType qui prendra le type assigné à la compilation. Le type compressedImage ne va pas stocker des pixels, mais des indexées qui seront associés à une table de correspondance qui stockera nos couleurs (Color Lookup Table ou C-LUT). Le type de données à utiliser pour notre image compressée est donc interchangeable selon nos besoins. De préférence, nous stockons les indexées sur le plus petit type pouvant stocker le nombre total de couleurs à utiliser (par exemple un GLubyte pour 256 couleurs, un Unsigned Short pour 512 couleurs).

3.1.b. Nuage de Couleurs

```

1 //Each color component
2 #define NB_COMPONENT 3
3 #define H 360

```

```

4 #define S 256
5 #define V 256
6
7 struct cloud{
8
9     short _data[H][S][V][NB_COMPONENT];
10 }
11 typedef struct cloud Cloud;

```

Comme dit dans l'introduction, BSP doit être appliqué sur l'espace des couleurs afin d'en réduire le nombre. Le voici. L'espace des couleurs peut être représenté tel un tableau à trois dimensions stockant chaque composante d'un pixel, nous l'appellerons nuage des couleurs.

Cependant, comme indiqué dans notre structure, nous n'utilisons pas les composants RGB (Red Green Blue), mais HSV (Hue Saturation Value). Le but de ce choix repose sur le fait que nous allons favoriser un espace où nous pouvons manipuler la saturation ou même la valeur afin d'avoir de meilleurs contrastes lors du rendu final. Effectivement, pour une compression égale, nous distinguons toujours mieux l'image en appliquant BSP sur un espace de couleurs HSV qu'un espace RGB. L'appliquer sur un espace RGB réduirait les contrastes mais favoriserait la fidélité des couleurs, tandis que l'appliquer sur un espace HSV augmenterait les contrastes, quitte à dégrader la réalité des couleurs. Vous l'aurez compris, nous voulons pouvoir distinguer l'image de préférence par ses formes et non par ses couleurs. Nous pouvons aussi noter que la dernière dimension de notre nuage correspond aux composantes Hue Saturation Value du pixel correspondant aux indexés H, S, V donnés.

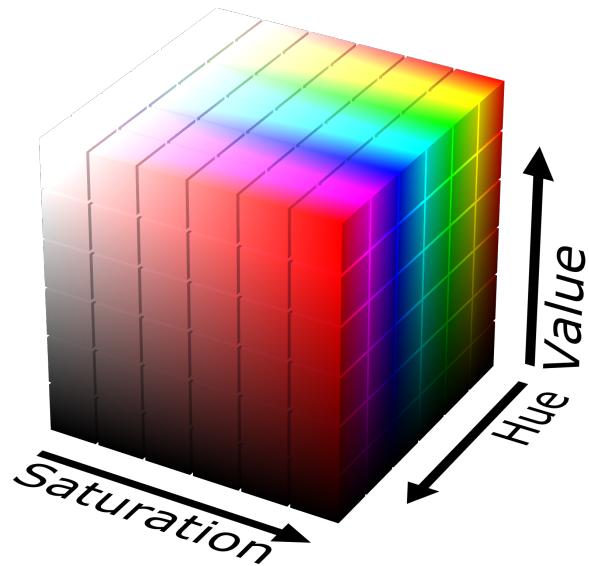


Figure 3: Une représentation de l'espace HSV

NB: Notons que la représentation de la Figure 3 est une représentation possible de HSV sous forme de solide et non celle décrite dans le code. Les axes ne sont pas disposés de la même manière et les dimensions ne sont pas les mêmes, mais le principe reste identique.

3.1.c. Représentation dans l'Espace

```

1 //8 Points to represent a rectangular space
2 #define NB_SUBSET_POINT 8
3 //4 Points to represent a cut plan

```

```

4 #define CUT_POINTS_SIZE 4
5
6 struct point{
7
8     short _x;
9     GLubyte _y;
10    GLubyte _z;
11};
12 typedef struct point Point;
13
14
15 struct cut{
16
17     Point _coordinates[CUT_POINTS_SIZE];
18 };
19 typedef struct cut Cut;
20
21
22 struct subset{
23     Point _coordinates[NB_SUBSET_POINT];
24 };
25 typedef struct subset Subset;

```

Afin de pouvoir travailler sur le nuage de couleurs, nous avons besoin de quelques outils afin de pouvoir diviser l'espace en plusieurs sous espaces. Nous utiliserons donc une structure point composée de 3 éléments pour les trois axes. Nous stockons la composante X d'un point sur un short et non un GLubyte car la longueur de notre nuage sur l'axe des X est de 360 donc supérieur à ce que peut accueillir un GLubyte (256).

Ensuite, nous avons une structure pour représenter une coupe afin de pouvoir sectionner notre espace par un plan. Un plan est composé de 4 points. Cette structure revient donc à utiliser un tableau de 4 points.

Pour finir, nous utiliserons une structure représentant les espaces et les sous espaces. Un espace ou un sous espace ici peut être représenté sous forme de parallélépipède, nous utiliserons donc un tableau de 8 points pour la représentation.

3.1.d. Arbre Binaire de Partitionnement de l'Espace

```

1 typedef struct node Node;
2 struct node{
3
4     Node* _leftChild;
5     Node* _rightChild;

```

```

6   Subset __subset;
7 }
```

Le principe de BSP est de découper chaque espace en deux nouveaux sous espace. C'est naturellement que nous allons utiliser un arbre binaire pour procéder au découpage de l'espace et ainsi stocker les sous espaces. Pour se faire, chaque noeud de l'arbre possède deux fils, et bien sûr, le sous espace qui lui est propre, utilisant la structure définie dans le point précédent. Précision tout de même que la racine de notre arbre binaire de partitionnement de l'espace comportera l'espace de notre nuage de point où nous allons travailler.

3.1.e. Table de Correspondance des Couleurs

```

1 typedef struct clutNode CLUTNode;
2 struct clutNode{
3
4     CLUTNode* __child;
5     GLubyte* __data;
6     usedType __index;
7 }
```

Nous reviendrons sur le principe de la C-LUT ultérieurement, mais nous pouvons tout d'abord comprendre le fonctionnement de sa structure de données. Cette structure est donc un noeud de la liste que nous générerons après. Chaque noeud de cette liste comporte bien entendu un enfant, un tableau de GLubyte permettant de stocker les composantes RGB d'une couleur, et un index pour cette même couleur.

Le choix de cette structure peut se justifier par une volonté d'ajouter des éléments au fur et à mesure, et non tous les éléments d'un coup. De plus, cela permettra de simplifier l'unicité des éléments qu'elle comporte car il se peut que nous puissions ajouter deux fois la même couleur, ce que nous ne voulons pas.

3.2. Partition Binaire de l'Espace

Notre nuage de point à pour dimensions 360x256x256 comme indiqué dans la partie précédente. Afin de représenter cet espace en donnée sur laquelle nous pouvons travailler, nous allons tout d'abord créer une structure "subset" à partir de ces dimensions données. Pour cela, nous utiliserons la fonction newSubsetFromDimensions().

```

1 Subset newSubsetFromDimensions(short firstDimensionSize, short secondDimensionSize, short
2   ↳ thirdDimensionSize) {
3
4     Subset toReturn;
5
6     firstDimensionSize--;
```

```

6     secondDimensionSize--;
7     thirdDimensionSize--;
8
9     toReturn._coordinates[0] = newPoint(0, 0, 0);
10    toReturn._coordinates[1] = newPoint(firstDimensionSize, 0, 0);
11    toReturn._coordinates[2] = newPoint(0, secondDimensionSize, 0);
12    toReturn._coordinates[3] = newPoint(firstDimensionSize, secondDimensionSize, 0);
13    toReturn._coordinates[4] = newPoint(0, 0, thirdDimensionSize);
14    toReturn._coordinates[5] = newPoint(firstDimensionSize, 0, thirdDimensionSize);
15    toReturn._coordinates[6] = newPoint(0, secondDimensionSize, thirdDimensionSize);
16    toReturn._coordinates[7] = newPoint(firstDimensionSize, secondDimensionSize, thirdDimensionSize);
17
18    return toReturn;
19 }
```

Une fois notre espace stocké, il faut définir une coupe vis à vis de ce dernier afin de pouvoir définir les deux sous espaces qui en découlent. Pour cela, utilisons la fonction `getCutFromSubset()` qui nous retournera un plan, donc 4 points, correspondant à une coupe de l'espace passé en argument. La fonction retournera toujours le plan coupant l'espace en deux, selon l'axe passé en argument (`X_AXE`, `Y_AXE`, `Z_AXE`).

Toutefois, nous pouvons noter que la formule utilisée dans cette fonction est:

Soit a et b deux points, et t le ratio:

$$t \in [0; 1]$$

$$x = (1 - t)a + tb$$

Elle a l'avantage de choisir un point selon une proportion donnée. Par défaut, nous utilisons une proportion de 50% pour définir un plan qui coupera l'espace en deux parties égales. Cependant, nous pouvons choisir de définir une coupe séparant l'espace de manière inégale, comme par exemple 30%/70%, si nous passons le ratio 30 en argument, ce qui fait la puissance de cette formule.

Définir une coupe qui est un plan ne coupant pas le nuage en deux parties équivalentes peut être intéressant selon le résultat voulu (plus de couleurs, plus de saturation...).

```

1 /*
2 Subset representation:
3   ^ 5-----6
4   Z /|      /|
5   / |  X>  /|
6   1-----2 |
7   | 7-----|-8
8   Y | /      | /
9   \/ /      |/
10  3-----4
11 */
12 //Numbers of axes
```

```

13 #define AXES_NB 3
14 //Macros to each axes
15 #define X_AXE 0
16 #define Y_AXE 1
17 #define Z_AXE 2
18
19 Cut getCutFromSubset(Subset * sub, int cutAxe, int pourcentage) {
20
21     //must affect them to avoid warnings --'
22     Point a = newPoint(0,0,0);
23     Point b = newPoint(0,0,0);
24     Point c = newPoint(0,0,0);
25     Point d = newPoint(0,0,0);
26
27     //According to x=(1-T)a+b formula to find a point on a segment in function of a ratio
28     //actually T is the ratio as T [0;1] & 2 points a, b.
29     float totalDistance, newPointDistance, T;
30
31     switch (cutAxe){
32         case X_AXE:
33
34             totalDistance = sub->_coordinates[1]._x - sub->_coordinates[0]._x;
35             newPointDistance = (totalDistance * pourcentage) / 100;
36
37             T = newPointDistance/totalDistance;
38
39             int x = (1 - T) * sub->_coordinates[0]._x + T * sub->_coordinates[1]._x;
40
41             a._x = x;
42             a._y = sub->_coordinates[0]._y;
43             a._z = sub->_coordinates[0]._z;
44
45             b._x = x;
46             b._y = sub->_coordinates[4]._y;
47             b._z = sub->_coordinates[4]._z;
48
49             c._x = x;
50             c._y = sub->_coordinates[2]._y;
51             c._z = sub->_coordinates[2]._z;
52
53             d._x = x;
54             d._y = sub->_coordinates[6]._y;
55             d._z = sub->_coordinates[6]._z;
56
57             break;
58
59         case Y_AXE:

```

```
60
61     totalDistance = sub->_coordinates[2]._y - sub->_coordinates[0]._y;
62     newPointDistance = (totalDistance * pourcentage) / 100;
63
64     T = newPointDistance/totalDistance;
65
66     int y = (1 - T) * sub->_coordinates[0]._y + T * sub->_coordinates[2]._y;
67
68     a._x = sub->_coordinates[0]._x;
69     a._y = y;
70     a._z = sub->_coordinates[0]._z;
71
72     b._x = sub->_coordinates[1]._x;
73     b._y = y;
74     b._z = sub->_coordinates[1]._z;
75
76     c._x = sub->_coordinates[4]._x;
77     c._y = y;
78     c._z = sub->_coordinates[4]._z;
79
80     d._x = sub->_coordinates[5]._x;
81     d._y = y;
82     d._z = sub->_coordinates[5]._z;
83
84     break;
85
86 case Z_AXE:
87
88     totalDistance = sub->_coordinates[4]._z - sub->_coordinates[0]._z;
89     newPointDistance = (totalDistance * pourcentage) / 100;
90
91     T = newPointDistance/totalDistance;
92
93     int z = (1 - T) * sub->_coordinates[0]._z + T * sub->_coordinates[4]._z;
94
95     a._x = sub->_coordinates[0]._x;
96     a._y = sub->_coordinates[0]._y;
97     a._z = z;
98
99     b._x = sub->_coordinates[1]._x;
100    b._y = sub->_coordinates[1]._y;
101    b._z = z;
102
103
104    c._x = sub->_coordinates[2]._x;
105    c._y = sub->_coordinates[2]._y;
106    c._z = z;
```

```

107
108     d._x = sub->_coordinates[3]._x;
109     d._y = sub->_coordinates[3]._y;
110     d._z = z;
111
112     break;
113
114     default:
115         break;
116     }
117
118     return newCutFromPoints(a, b, c, d);
119 }
```

Maintenant que nous avons vu les éléments fondamentaux pour générer notre arbre, passons à sa création. Rappelons tout d'abord le principe de notre arbre binaire qui va permettre de stocker tous les espaces dont nous avons besoin.

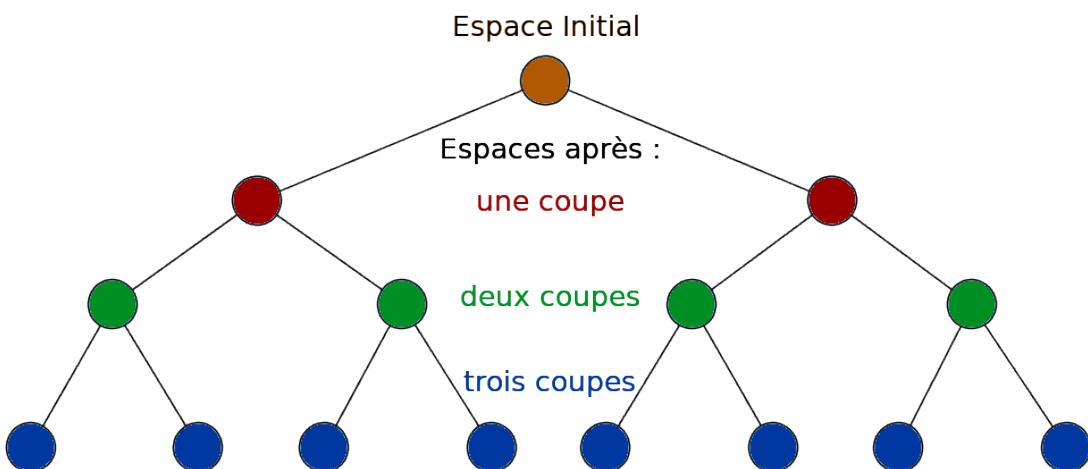


Figure 4: Representation de notre arbre BSP

Comme la Figure 4 nous l'indique, la racine de l'arbre est notre espace initiale, donc un espace possédant les dimensions de notre nuage de points. De cet espace découle deux sous espaces. De ces deux sous espaces, quatre sous espaces et ainsi de suite...

Ci dessous, la fonction de conception de l'arbre binaire que nous allons utiliser:

```

1 //Numbers of axes
2 #define AXES_NB 3
3
4 void createTree(Node * n, int cpt, Cut cut, int actualCutAxe) {
5
6     if(cpt <= 0){
```

```

7     return;
8 }
9
10 Subset left;
11 Subset right;
12
13 int nextCut = (actualCutAxe+1)%AXES_NB;
14
15 switch (actualCutAxe){
16     case X_AXE:
17
18         left = newSubsetFromPoints(n->_subset._coordinates[0],cut._coordinates[0],n-
19             ->_subset._coordinates[2],cut._coordinates[2],
20             ->_subset._coordinates[4],cut._coordinates[1],n-
21             ->_subset._coordinates[6],cut._coordinates[3]);
22
23         right = newSubsetFromPoints(cut._coordinates[0],n-
24             ->_subset._coordinates[1],cut._coordinates[2],n->_subset._coordinates[3],
25             ->_subset._coordinates[5],cut._coordinates[3],n-
26             ->_subset._coordinates[7]);
27
28         break;
29
30     case Y_AXE:
31
32         left = newSubsetFromPoints(n->_subset._coordinates[0],n-
33             ->_subset._coordinates[1],cut._coordinates[0],cut._coordinates[1],
34             ->_subset._coordinates[4],n-
35             ->_subset._coordinates[5],cut._coordinates[2],cut._coordinates[3]);
36
37         right = newSubsetFrom-
38             Points(cut._coordinates[0],cut._coordinates[1],cut._coordinates[2],cut._coordinates[3],
39             ->_subset._coordinates[4],n->_subset._coordinates[5],n->_subset._coordinates[6],n-
40             ->_subset._coordinates[7]);

```

```

37         break;
38
39     default:
40         break;
41     }
42
43
44     addLeftNode(left,n);
45     createTree(n->_leftChild,cpt-1,getCutFromSubset(&left, nextCut, 50),nextCut);
46     addRightNode(right,n);
47     createTree(n->_rightChild,cpt-1,getCutFromSubset(&right, nextCut, 50),nextCut);
48 }
```

C'est une fonction récursive de création d'arbre binaire de tout ce qui a de plus classique, hormis l'utilisation de nouveaux espaces lors de la création des noeuds selon l'axe utilisé. Essayons de comprendre son fonctionnement.

Tout d'abord, la fonction prend en argument un noeud. Lors de l'appel initial, ce sera la racine qu'il faudra lui passer, c'est à dire, notre espace représentant notre nuage. De même, il faudra passer à la fonction la coupe préalablement réalisée sur l'espace initial. L'argument "cpt" quant à lui représente la profondeur de notre arbre binaire. L'arbre binaire étant complet, nous pouvons en conclure que son nombre de feuilles est de $2^{profondeur}$. Son nombre de feuilles correspond au nombre de sous-espaces totaux.

La fonction doit créer selon l'axe de la coupe actuelle (passé en argument) deux sous espaces à partir de l'espace passé en argument. Selon chaque axe de coupe, la manière de créer les deux sous espaces sera différente. Nous sommes obligés de faire du cas par cas.

De manière plus générale, nous pouvons dire que nous assignons la face de gauche de l'espace initial à notre nouvel espace pour le fils gauche, et que nous assignons la coupe comme face de droite pour l'espace du fils droit. Vice versa pour l'espace du fils droit.

Une fois nos deux nouveaux espaces créés, nous rappelons logiquement notre fonction pour chaque fils. Nous l'appelons en décrémentant le compteur de 1, en générant une nouvelle coupe à partir de l'espace en question, et en utilisant un axe de coupe différent de l'ancien.

3.3. Principe de la C-LUT

C-LUT ou en anglais Color Lookup Table fait référence à une table de correspondance de couleur. Son principe est assez simple. Chaque couleur de la table correspond à un index, et donc par réciproque, chaque index correspond à une couleur dans la table. L'utilité de la C-LUT dans notre compression repose sur un principe logique: Un index prend moins de mémoire qu'une couleur à trois composantes RGB.

Comme dit précédemment, nous ne voulons pas de doublons lorsque nous allons stocker nos

couleurs dans la table. Pour se faire, nous allons vérifier si la couleur n'est pas déjà inscrite dans notre C-LUT lors de l'ajout d'une couleur. Voici la fonction d'ajout d'élément à notre table, et par conséquent l'affectation d'un nouveau noeud:

```

1 void setCLUTNode(GLubyte* data, usedType index, CLUTNode * n){
2
3     int i;
4
5     n->_data = (GLubyte*)malloc((size_t) 3 * sizeof(GLubyte));
6
7     for(i = 0; i< 3; ++i){
8
9         n->_data[i] = data[i];
10    }
11
12    n->_index = index;
13 }
14
15 void addCLUTNodeChild(GLubyte* data, CLUTNode* father){
16
17     while(father->_child != NULL){
18
19         if(father->_child->_data[0] == data[0] && father->_child->_data[1] == data[1] &&
20             father->_child->_data[2] == data[2]){ //if the color is already in our CLUT
21
22             return;
23         }
24         father = father->_child;
25     }
26
27     CLUTNode* n = newEmptyCLUTNode();
28     setCLUTNode(data, father->_index + 1, n);
29
30     father->_child = n;
31 }
```

Fonctions assez triviales en soit. Notons tout de même l'ajout de l'index correspondant à la couleur stockée, dont nous aurons besoin par la suite. Notons que lors de l'ajout d'un noeud à la liste, nous vérifions en préambule si la valeur de la couleur à ajouter n'est pas identique à une couleur qui est déjà dans la table. Dans ce cas là, on sort directement de la fonction, nul besoin de continuer le parcours de la liste.

3.4. Sélection des Nouvelles Couleurs

Nous avons à présent une panoplie complète d'outils pour modifier notre nuage de couleurs ainsi que notre C-LUT. Pour ce faire, nous allons parcourir notre arbre BSP afin

de se positionner sur chaque feuille de l'arbre, qui correspond à un espace. Chaque espace représente donc une partie du nuage de couleur.

```

1 void modifyCloudFromTree(Cloud * c, CLUTNode * CLUT, Node * tree){
2
3     if(tree->_leftChild != NULL){
4         modifyCloudFromTree(c, CLUT, tree->_leftChild);
5     }
6
7     if(tree->_rightChild != NULL){
8         modifyCloudFromTree(c, CLUT, tree->_rightChild);
9     }
10
11    if(tree->_leftChild == NULL && tree->_rightChild == NULL) { //leaf -> subset
12
13        /*Calculus for the average of all components of the subset*/
14        int i, j, k;
15        int r, g, b;
16        long long int hAverage = 0, sAverage = 0, vAverage = 0;
17        long long int nbIterations = ((tree->_subset._coordinates[7]._x -
18            tree->_subset._coordinates[0]._x)
19                * (tree->_subset._coordinates[7]._y - tree->_subset._coordinates[0]._y)
20                * (tree->_subset._coordinates[7]._z - tree->_subset._coordinates[0]._z));
21
22        GLubyte* CLUTData = (GLubyte*)malloc((size_t) 3 * sizeof(GLubyte));
23
24        for(i = tree->_subset._coordinates[0]._x; i < tree->_subset._coordinates[7]._x; ++i){
25
26            for(j = tree->_subset._coordinates[0]._y; j < tree->_subset._coordinates[7]._y; ++j){
27
28                for(k = tree->_subset._coordinates[0]._z; k < tree->_subset._coordinates[7]._z; ++k){
29
30                    hAverage+= i;
31                    sAverage+= j;
32                    vAverage+= k;
33                }
34            }
35
36            hAverage/=nbIterations;
37            sAverage/=nbIterations;
38            vAverage/=nbIterations;
39
40
41            /*Assignment of the subset average component for every components of the subset*/
42            for(i = tree->_subset._coordinates[0]._x; i < tree->_subset._coordinates[7]._x; ++i){
43
44                for(j = tree->_subset._coordinates[0]._y; j < tree->_subset._coordinates[7]._y; ++j){

```

```

45
46     for(k = tree->_subset._coordinates[0]._z; k < tree->_subset._coordinates[7]._z; ++k){
47
48         c->_data[i][j][k][0] = (short)hAverage;
49         c->_data[i][j][k][1] = (short)sAverage;
50         c->_data[i][j][k][2] = (short)vAverage;
51     }
52 }
53 }

54
55 hsv2rgb(hAverage, sAverage, vAverage, &r, &g, &b);
56 /*ADDING COLOR TO OUR CLUT*/
57 CLUTData[0] = (GLubyte)r;
58 CLUTData[1] = (GLubyte)g;
59 CLUTData[2] = (GLubyte)b;

60
61 addCLUTNodeChild(CLUTData, CLUT);

62
63 free(CLUTData);

64
65 }
66 }
```

Une fois positionné sur un espace, il suffit de faire la moyenne de toutes les couleurs de cet espace. Ensuite, on affecte la couleur moyenne de l'espace à la partie concernée de notre nuage de couleur. Une fois cela fait, on pourra accéder à notre nouvelle couleur (la couleur moyenne) en passant les index de la couleur originale à notre nuage de couleur.

De plus, la couleur moyenne par espace nous intéresse pour notre C-LUT. On l'ajoute alors en convertissant sa valeur en format RGB car c'est ce que nous utiliserons lors de l'écriture de notre C-LUT dans le fichier compressé.

3.5. Création de l'Image Compressée

Une fois notre C-LUT rempli ainsi que notre nuage de couleur modifié, il ne reste plus qu'à transformer notre image de base de type "Image" en type "CompressedImage". Pour cela, nous devons faire appel à la fonction `newCompressedImageFromCloud()` qui nous retournera la nouvelle image compressée. Elle sera remplie d'index faisant référence à la C-LUT et non à des couleurs à proprement parlé.

```

1 CompressedImage newCompressedImageFromCloud(Cloud * c, Image * img, CLUTNode * root){
2
3     int i, h = 0, s = 0, v = 0, r = 0, g = 0, b = 0;
4     CompressedImage new;
5     new.sizeX = img->sizeX;
6     new.sizeY = img->sizeY;
```

```

7   int size = new.sizeX * new.sizeY;
8   new.data = (usedType *) malloc ((size_t) size * sizeof (usedType));
9
10  GLubyte* CLUTData = (GLubyte*)malloc((size_t) 3 * sizeof(GLubyte));
11
12  for(i = 0; i < size * 3; i += 3 ){
13
14      rgb2hsv(img->data[i], img->data[i+1], img->data[i+2], &h, &s, &v);
15
16      hsv2rgb(c->_data[h][s][v][0], c->_data[h][s][v][1], c->_data[h][s][v][2], &r, &g, &b);
17
18
19      CLUTData[0] = r;
20      CLUTData[1] = g;
21      CLUTData[2] = b;
22
23
24      new.data[(int)i/3] = getIndexFromData(CLUTData, root);
25  }
26  free(CLUTData);
27
28  return new;
29 }
```

Dans cette fonction, nous copions d'abord les dimensions de l'image originale dans notre nouvelle image compressée. Ensuite, pour chaque pixel de l'image originale, nous convertissons sa couleur avec des composantes Rouge Vertes Bleu en Hue Saturation Value (ce qui est utilisé dans notre nuage de couleur). Possédant maintenant la couleur du pixel sous forme HSV, il est très simple de récupérer la couleur définie lors de la phase de sélection des nouvelles couleurs. On obtient notre nouvelle couleur en accédant à notre nuage de couleurs par les composantes HSV de la couleur originale comme index.

Pour finir, on assigne à notre image compressée l'index de chaque couleur correspondante dans la C-LUT par la fonction `getIndexFromData()`, qui retourne l'index d'une couleur selon ses composantes.

NB: Notons que nous possédons dans notre code également une fonction `newImageFromCloud()` qui permet de créer une image non compressée, mais avec les couleurs que l'image compressée prendrait. Elle sert donc à créer une image de prévisualisation à afficher avant une potentielle compression.

3.6. Ecriture et Lecture de l'Image Compressée

3.6.a. Ecriture

Actuellement, nous nous retrouvons avec notre image compressée dans la mémoire vive de notre machine, mais nous aimerais la stocker de manière permanente dans notre mémoire morte. Possédant notre nouvelle image compressée ainsi que notre C-LUT, rien de plus simple: Il suffit d'écrire de manière binaire ces informations dans un fichier que nous créerons. Cependant, nous allons essayer d'optimiser la place que nous prenons lors de l'écriture de nos données. Pour cela, nous allons utiliser le type plus gros que le C possède, c'est à dire un Long Int. Le principe est que nous allons stocker le maximum d'index par Long Int afin de ne pas gâcher l'espace d'un type dont nous n'exploitons pas toute la mémoire. Nous travaillerons donc aux niveaux des bits de ce type afin d'optimiser la place que nous prenons par index.

```

1 void compressToBSP(char *filename, CompressedImage *img, CLUTNode* root){

2
3     FILE *fp;
4     //open file for output
5     fp = fopen(filename, "wb");
6     if (!fp) {
7         fprintf(stderr, "Unable to open file '%s'\n", filename);
8         exit(1);
9     }

10    int cpt = 0;
11    char type;

12
13    if(treeDepth <= 8){
14        type = 0;
15    } else {
16        type = 1;
17    }

18
19    fwrite(&type, (size_t) 1, sizeof(char), fp);

20
21    if(treeDepth <= 8){
22        //writing the number of subset of GLubyte
23        GLubyte nbSubset = (GLubyte)getCLUTSize(root);
24        fwrite(&nbSubset, (size_t) 1, sizeof(GLubyte), fp);
25    } else {
26        //writing the number of subset as ushort
27        unsigned short nbSubset = (unsigned short)getCLUTSize(root);
28        fwrite(&nbSubset, (size_t) 1, sizeof(unsigned short), fp);
29    }

30
31
32    //image size
33    fwrite(&img->sizeX, (size_t) 1, sizeof(unsigned long), fp);

```

```

34     fwrite(&img->sizeY, (size_t) 1, sizeof(unsigned long), fp);
35
36     // clut
37     CLUTfileWriter(fp, root->_child, type);
38
39
40     char * binary= (char*)malloc(treeDepth * sizeof(char));
41
42     // pixel data
43     while(cpt < (int)(img->sizeY * img->sizeX)){
44
45         unsigned long int dataToWrite = 0x00;
46
47         for(int i = 0; i < (int)(floor(64/treeDepth)); ++i){
48
49             decimalToBinary(img->data[cpt], &binary);
50
51             for(int j = treeDepth - 1; j>=0; --j){
52
53                 if(binary[j] == 1){
54                     dataToWrite |= 1UL << ((i * treeDepth) + j);
55                 }
56             }
57             ++cpt;
58         }
59
60         fwrite(&dataToWrite, (size_t) 1, (size_t)(sizeof(unsigned long int)), fp);
61
62     }
63
64     free(binary);
65
66     fclose(fp);
67 }
```

Dans un premier temps, nous inscrivons dans notre fichier le type utilisé pour stocker les index de la C-LUT. Il est utile de l'indiquer car lors de la lecture du fichier, le programme pourra s'adapter au type utilisé. Ensuite, on ajoute le nombre de couleurs que nous allons utiliser, en le récupérant par la fonction getCLUTSize() et on écrit aussi les dimensions de l'image.

La fonction CLUTfileWriter() est applée après. C'est une fonction qui parcourera notre C-LUT, et qui inscrira pour chaque élément, 4 valeurs:

- L'index
- La composante Rouge de la couleur
- La composante Verte de la couleur

- La composante Bleu de la couleur

```

1 void CLUTfileWriter(FILE *fp, CLUTNode* n, char type){
2
3     if(n->_child != NULL){
4
5         CLUTfileWriter(fp, n->_child, type);
6     }
7
8     GLubyte r, g, b;
9     r = n->_data[0];
10    g = n->_data[1];
11    b = n->_data[2];
12
13    if(type == 0){ //GLubyte
14
15        GLubyte index;
16        index = (GLubyte)n->_index;
17
18        fwrite(&index, (size_t) 1, sizeof(GLubyte), fp);
19    } else { //ushort
20
21        unsigned short index;
22        index = (unsigned short)n->_index;
23
24        fwrite(&index, (size_t) 1, sizeof(unsigned short), fp);
25    }
26    fwrite(&r, (size_t) 1, sizeof(GLubyte), fp);
27    fwrite(&g, (size_t) 1, sizeof(GLubyte), fp);
28    fwrite(&b, (size_t) 1, sizeof(GLubyte), fp);
29 }
```

Une fois notre C-LUT écrite, il nous reste plus qu'à inscrire les index correspondant à la couleur de chaque pixel de l'image. Pour cela, nous devons déjà définir sur combien de bits nous allons stocker un index. Utilisant un arbre binaire, rien de plus simple: la profondeur à laquelle nous appelons BSP correspondra au nombre de bits à utiliser pour stocker l'index maximal. Visuellement, il est facile de comprendre pourquoi en reprennant la Figure 4 représentant notre arbre.

Ensuite, pour assigner nos différents index sur le même Long Int, il faudra définir combien d'index peuvent loger sur ce type sans dépassement. Il suffit d'appliquer $\text{floor}(64/\text{treeDepth})$, qui est finalement la partie entière d'une division Euclidienne pour avoir le nombre d'éléments logeant sur ce type (64 étant le nombre de bits que comporte un Long Int).

```

1 for(int i = 0; i < (int)(floor(64/treeDepth)); ++i){
```

```

3     decimalToBinary(img->data[cpt], &binary);
4
5     for(int j = treeDepth - 1; j>=0; --j){
6
7         if(binary[j] == 1){
8             dataToWrite |= 1UL << ((i * treeDepth) + j);
9         }
10    }
11    ++cpt;
12 }

```

Afin de transformer nos index sous forme de chiffres binaires, nous faisons appel à la fonction decimalToBinary() qui se chargera de transformer nos index décimaux sous forme de tableaux de "bits" (autrement dit un nombre binaire) d'une longueur égale à la profondeur de notre BSP. Possèdant un Long Int initialisé à 0 sur tous ses bits, il ne nous reste plus qu'à écrire uniquement les bits valant 1, pour chaque index, par l'opérateur binaire OR (en prenant soin de se positionner sur le bit correspondant dans notre Long Int). Par ce biais, nous limitons l'espace inutilisé lors de l'écriture fichier !

3.6.b. Lecture

Afin de retranscrire nos données de l'image en données PPM P6 (format de l'image initial), nous devons récupérer les données préalablement écrites afin de remplir une structure "Image". Pour cela, nous utiliserons la fonction loadCompressedBSP(). Le principe de la fonction est simple. Ouvrir le fichier où nous avons stocké les données de l'image compressée de manière binaire, et remplir une strucutre image. Ici, nous présenterons seulement la partie qui remplira notre nouvelle structure image par les couleurs correspondantes aux index stockés dans des GLubytes (la récupération de données n'étant pas intéressante, et ayant le même principe pour des unsigned shorts...).

```

1 i = 0;
2 while ( i < (int)(toReturn.sizeX * toReturn.sizeY)){
3
4     fread(&binarysData, sizeof(long int), 1, fp);
5
6     for(int j = 0; j < (int)(floor(64/treeDepth)); ++j){
7         for(int k = treeDepth - 1; k>=0; --k){
8
9             binary[k] = (char)(binarysData >> ((j * treeDepth) + k)) & 0x01;
10        }
11
12        index = (GLubyte)binaryToDecimal(&binary);
13
14        toReturn.data[(int)i*3] = r[index];
15        toReturn.data[(int)(i*3)+1] = g[index];
16        toReturn.data[(int)(i*3)+2] = b[index];

```

```

17
18     i++;
19 }
20 }
```

De la même manière dont nous avons parcouru nos Long Int lors de la phase d'écriture, nous devons récupérer nos données stockées sous formes de bits. Pour cela, nous récupérons le tableau de bit correspondant (chiffre binaire) pour chaque index, par le biais de l'opérateur ET binaire. Afin de transformer notre index en décimal, nous faisons appel à la fonction `binaryToDecimal()` qui se chargera de transformer l'index sous une forme utilisable.

Une fois cette fonction appliquée, il ne restera plus qu'à utiliser la fonction `imageSavePPM()` afin de sérialiser les données de l'image sous forme de fichier PPM P6.

4. Analyse

4.1. Taux de Compressions

Une image sous format PPM P6 stocke chaque couleur d'un pixel sous la forme d'un triplet de GLubyte RGB (unsigned char). En omettant l'entête de l'image PPM, le poids de cette image correspond à:

Soit np le nombre de pixel de l'image et tg la taille d'un GLubyte:

$$np * tg * 3$$

Cependant, nous stockons ces informations de manière différente dans notre fichier compressé. On inscrit tout d'abord la C-LUT qui aura donc un poids de:

Soit ti la taille du type utilisé pour stocker un index (GLubyte ou Unsigned Short) et nc le nombre de couleurs de la C-LUT:

$$(ti * nc) + (tg * nc * 3)$$

On inscrit logiquement nos index correspondants à une couleur dans la C-LUT pour chaque pixel de l'image. Pour cela, nous utiliserons des Long Int pour stocker nos index. La formule sera donc:

Soit np le nombre de pixel de l'image et d la profondeur utilisée pour notre BSP:

$$((ti * nc) + (tg * nc * 3)) + (np / \lfloor (64 / d) \rfloor) * 64$$

Comparons maintenant le poids d'une image PPM P6 avec des compressions utilisant 64

couleurs, 256 couleurs, et 2048 couleurs.

Soit une image PPM P6 de dimension 1024x768. L'image comporte donc 786432 pixels. Un GLubyte ayant un poids de 8 bits nous pouvons facilement en déduire que le poids de l'image en bits correspond à :

$$786432 * 8 * 3 = 18874368 \text{ bits}$$

Soit 2359,296 kiloctets

Calculons maintenant le poids de l'image pour une compression avec 64 couleurs:

$$((8 * 64) + (8 * 64 * 3)) + (786432 / \lfloor (64 / 6) \rfloor) * 64 = 5035212 \text{ bits}$$

Soit 629,4015 kiloctets

Pour 256 couleurs:

$$((8 * 256) + (8 * 256 * 3)) + (786432 / \lfloor (64 / 8) \rfloor) * 64 = 6299648 \text{ bits}$$

Soit 787,456 kiloctets

Et pour finir 2048 couleurs:

$$((16 * 2048) + (8 * 2048 * 3)) + (786432 / \lfloor (64 / 11) \rfloor) * 64 = 10131865 \text{ bits}$$

Soit 1266,483125 kiloctets

Pour la compression par BSP, le taux de compression d'une image se calcul donc par:

$$(100 / 3) * (d * (8 / 64)) \%$$

Nous pouvons donc en conclure que la compression de l'image en terme de poids dépend essentiellement du nombre de couleurs que contient notre C-LUT. En effet, plus on peut stoquer nos index sur un type petit, plus nous réduisons la taille de l'espace utilisé.

4.2. Perte de Qualité

Bien entendu, pour ce qui est de la perte de qualité, nous passons à nombre de couleurs drastiquement réduit. Il y en aura donc forcément de la perte de qualité. Cependant plusieurs facteurs peuvent jouer sur cette perte de qualité.

Avant toute chose, voici un sondage rempli par plusieurs personnes afin de juger de la perte de qualité d'une image, selon les options utilisées (première coupe à différents endroits, changer l'axe de coupe, augmenter le nombre de couleurs à utiliser...):

- [Sondage](#)
- [Réponses](#)

De mon point de vue, le facteur de compression le plus important est le nombre de couleurs utilisées dans l'image. Comparons une image compressée utilisant 256 couleurs et une autre utilisant 65 536 couleurs. La différence est frappante.

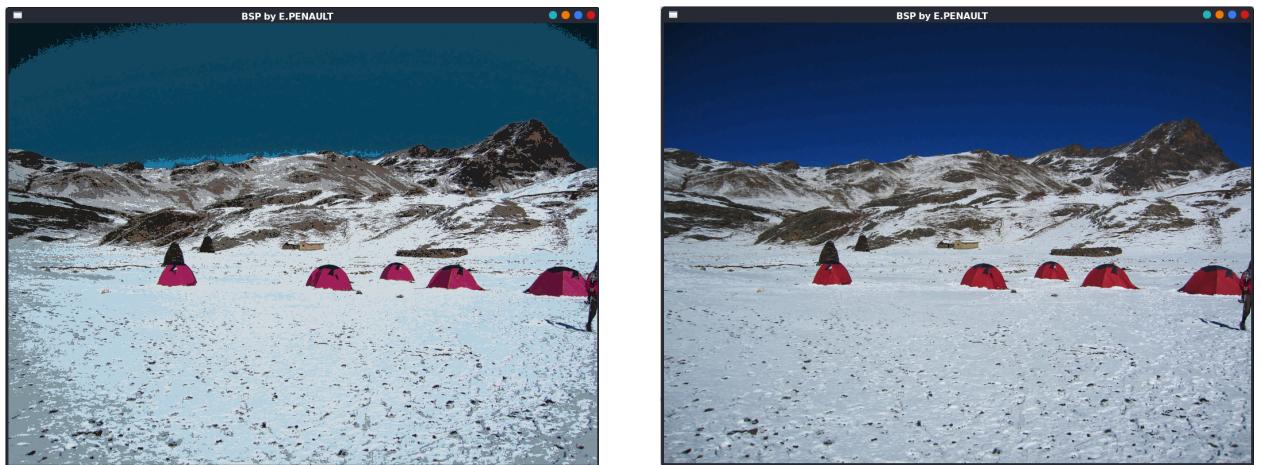


Figure 5: Comparaison de deux images, l'une comportant 256 couleurs, l'autre 65 536

Plus nous utilisons de couleurs, moins l'image sera dégradée. Avec 65 536 couleurs, il peut parfois être difficile de voir qu'une compression a été appliquée.

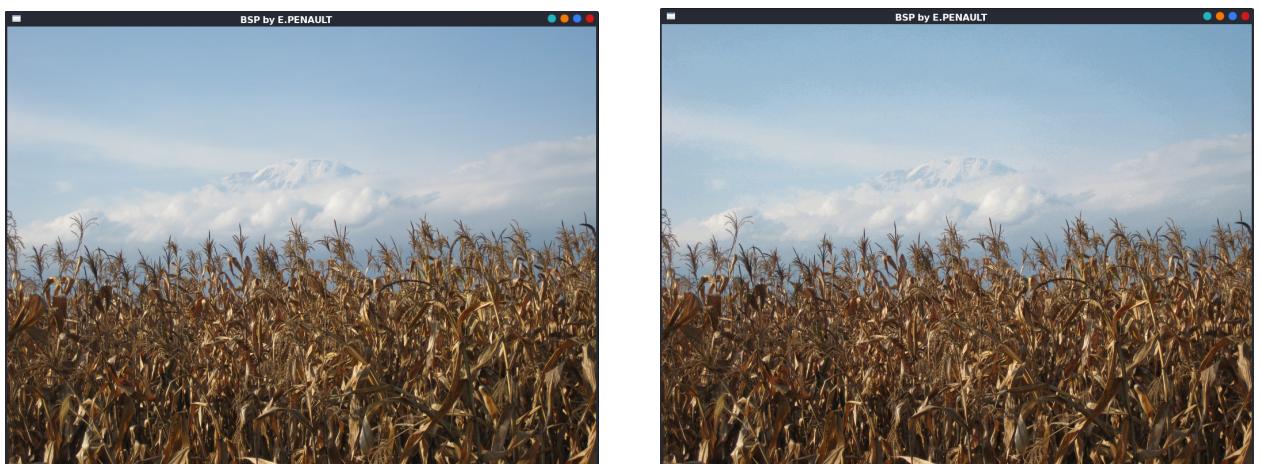


Figure 6: L'image originale à gauche, celle compressée avec 65 536 couleurs à droite

Un facteur qui peut jouer sur le visuel de l'image est le placement de la coupe initial:

On peut voir sur la Figure 7 que l'importance du placement de la première coupe est cruciale. Cela permet d'accentuer une partie de la teinte souhaitée, augmenter la saturation de l'image, ou même augmenter des contrastes.

Nous pouvons voir que les deux images diffèrent entre elles par la couleur des tentes. Cependant nous remarquons que peu importe l'image, le ciel est plus foncé qu'une coupe divisant l'espace en deux parties égales (CF: première image de la Figure 5).

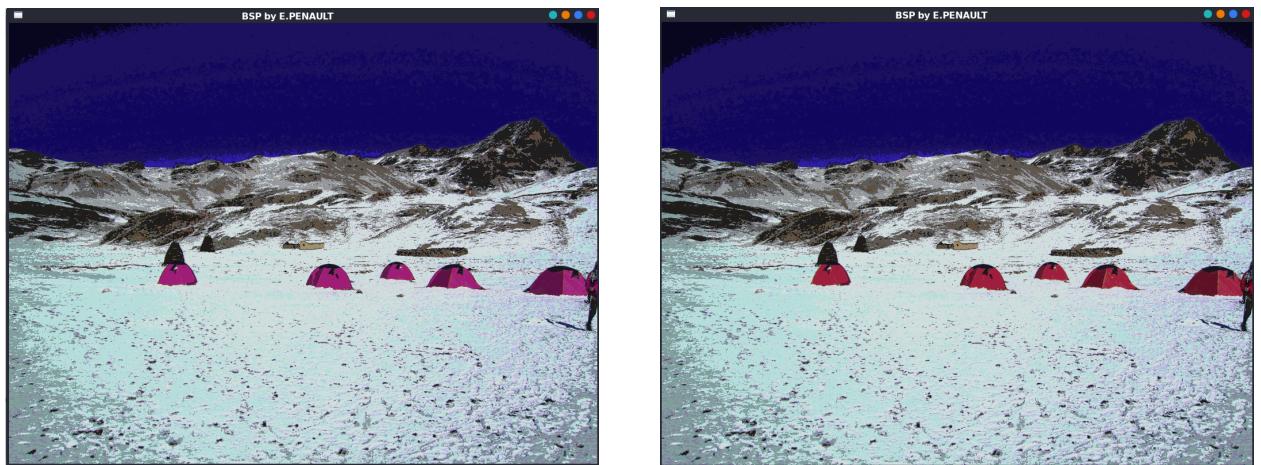


Figure 7: Deux images avec une même coupe, mais réalisée à des niveaux différents (20% et 80%)

Pour finir, un autre facteur qui peut être intéressant est l'axe de la première coupe de notre nuage de couleur. Le changer peut amener à des résultats différents qui n'en reste pas moins intéressants:



Figure 8: Trois images ayant respectivement une première coupe en X, Y et Z

On peut voir ici que selon la première coupe réalisée dans notre espace de couleurs, les couleurs obtenues sont différentes. Nous pouvons y trouver un intérêt si nous voulons conserver une teinte de couleurs en particulier, et par conséquent en dégrader une autre (comme nous pouvons le voir sur la seconde image ci-dessus, la couleur du ciel est plus fidèle à l'image originale, mais on dégrade alors la couleur des tentes...).

4.3. Temps d'Exécution

Le dernier facteur qui nous intéresse pour la compression est le temps que met le programme à compresser une image.

Voici un tableau des temps en fonction de la profondeur de l'arbre BSP utilisée (L'écriture et la lecture étant instantanées, il n'est pas utile de la comparer ici). Nous comparerons les temps trouvés afin d'analyser l'évolution de ces derniers:

Profondeur de l'arbre BSP	Temps en secondes
1	0.078239 sec
2	0.103695 sec
3	0,094425 sec
4	0.093786 sec
5	0.113167 sec
6	0.126987 sec
7	0.173623 sec
8	0.255451 sec
9	0.423419 sec
10	1.193781 sec
11	2.364104 sec
12	4.563384 sec
13	9.073874 sec
14	17.965867 sec
15	37.366011 sec
16	71.030565 sec

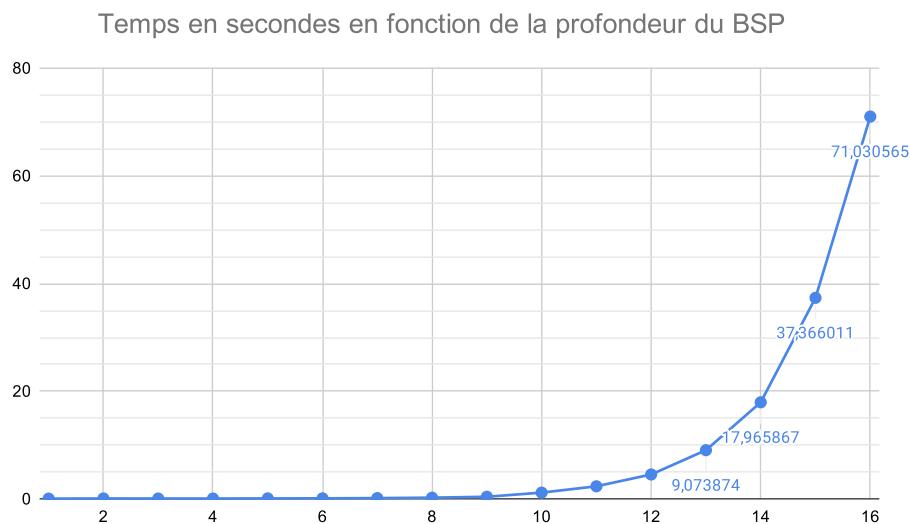


Figure 9: Graphique du temps en secondes en fonction de la profondeur du BSP

Visuellement nos résultats de temps sont très parlant. L'allure de la courbe prend une allure exponentielle. Si nous regardons les résultats de plus près, nous nous rendons compte que le temps double plus ou moins à chaque fois que nous augmentons la profondeur d'une unité. Plus la profondeur utilisée pour notre arbre BSP est conséquente, plus nous mettrons de temps à générer notre image compressée.

Notons tout de même que la fonction `newCompressedImageFromCloud()` prend presque toute la totalité du temps indiqué ci-contre. Cette fonction doit parcourir toute notre liste représentant notre C-LUT pour chaque pixel afin de trouver l'index de la couleur correspondante à la couleur du pixel actuel. Naturellement, plus nous aurons de couleurs dans notre C-LUT, plus nous mettrons de temps.

Une option pour optimiser le temps de recherche d'un index dans la C-LUT aurait été d'implémenter une structure "set" avec un Red–Black Tree, comme est implémentée la structure standarde "set" en C++.

5. Difficultés

Je n'ai pas eu de difficultés particulières lors de ce projet à proprement parlé, ici ce n'est la compréhension du sujet. Avant de travailler sur l'espace des couleurs, je travaillais sur l'image même, ce qui m'a valu une perte de temps... De plus, j'ai eu quelques soucis avec mes fonctions de conversion entre le formats RGB et HSV, ce qui m'a rendu la tache légèrement plus complexe.

6. Conclusion

Pour conclure, nous pouvons dire de manière sûre que la compression par BSP est une compression avec perte. Cependant, cette perte semble dépendre de plusieurs facteurs et nous pouvons la limiter. Pour la limiter, nous pouvons faire un compromis sur le temps et le taux de compression en augmentant le nombre de couleurs que nous allons utiliser. De plus, nous pouvons jouer sur l'axe des coupes ainsi que leurs emplacements afin d'accentuer certains traits de l'image sans pour autant modifier le nombre de couleurs choisies et donc, ne pas impacter le temps de compression.

Je juge le résultat du projet plutôt convenable même s'il pourrait être amélioré (comme l'implémentation d'une strucutre "set" pour réduire le temps d'accès aux index de ma C-LUT, ou utiliser une matrice compacte pour représenter mon nuage de couleurs...)

7. Remerciements

Mes remerciements à:

- Jean-Jaques BOURDIN pour m'avoir aiguillé dans ce projet.
- Angela PECURICA pour l'aide matérielle apportée avec un Mac.

8. Sources

- Figure 3: https://commons.wikimedia.org/wiki/File:HSV_color_solid_cube.png