Traitement optimisé de la couleur, tramage

Michel Buffa, UNSA, UFR Sciences, 1996

1 Introduction

Dans ce chapitre nous allons essayer de répondre à divers problèmes qui se posent lorsqu'on manipule de la couleur en synthèse d'images. Supposons que l'on ne dispose que d'un écran permettant d'afficher simultanément *n* couleurs (le plus souvent 256) parmi une palette beaucoup plus large (16 millions de couleurs).

- Lorsque plusieurs applications tournent simultanément, comment afficher les couleurs de son choix sans en voler à l'application voisine ?
- Si on ne dispose que de *n* couleurs, comment afficher de manière optimale une image, une photo ou simuler un fin dégradé qui requièrent pourtant plusieurs milliers de couleurs ?

Nous allons étudier deux techniques complémentaires pour remédier à ces problèmes. La première s'appuie sur une optimisation de la table des couleurs disponibles (colormap), l'autre consiste à tramer les couleurs pour simuler les couleurs manquantes. Nous verrons en particulier le célèbre tramage de Floyd-Steinberg (*Floyd-Steinberg diffusion dithering*), qui est très simple à implanter et donne des résultats fantastiques lorsqu'elle est associée avec une table de couleurs optimisée.

2 Median Cut Color Quantization

La quantization de couleurs est une technique qui consiste à choisir les meilleures *n* couleurs affichables simultanément parmi la palette, pour afficher une image contenant des milliers ou des millions de couleurs. En particulier, nous allons étudier la *median cut color quantization*, développée en 1982 par Paul Heckbert au New York Institute of Technology. Cette technique est largement utilisée aujourd'hui par les programmes de traitement d'image (photoshop, etc...) et autres programmes graphiques.

Supposons que nous faisons tourner notre application sur une machine permettant d'afficher 256 couleurs. On désire afficher une image 24 bits contenant des milliers de couleurs différentes (le terme 24 bits signifie que la couleur de chaque pixel est représentée par une valeurs codée sur 24 bits, 8 bits pour chaque composante RVB, soit 256 niveaux de rouge, de vert et de bleu pour décrire 16 millions de nuances). Cette image contient de nombreuses et subtiles nuances de bleu, de jaune et de vert, mais juste un peu de rouge (un coquelicot perdu dans un champ de pissenlits, superbe ciel bleu). Une bonne idée avant d'afficher l'image est de choisir les 256 couleurs qui vont permettront un rendu optimal. Mais comment choisir les "meilleures couleurs" ?

Une approche consiste à programmer la table de couleurs avec un arc-en-ciel de couleurs représentant le blanc, le noir, et un éventail de couleurs variées entre les deux. Avec 256 couleurs à notre disposition, on pourrait créer une table de couleurs contenant 16 nuances de 16 couleurs différentes. Cette méthode, très simple (elle ne nécessite que très peu de ligne de code et permet un traitement très rapide), est fréquemment utilisée. Son principal avantage (à part la simplicité), est qu'elle garantit qu'il existe toujours dans la table une couleur proche de chaque couleur présente dans l'image. L'inconvénient est que l'image a tendance à perdre certaines nuances subtiles. Une centaine de variations de bleu par exemple, peuvent être converties en seulement trois ou quatre nuances. Une magnifique ciel bleu allant du bleu azuré au bleu nuit sera affiché sous forme de larges bandes sans nuances.

Une autre approche souvent utilisée consiste à compter le nombre de couleurs dans l'image et le nombre d'apparitions de chaque couleur, puis à programmer la table de couleurs avec les 256 couleurs les plus fréquentes. Ainsi, on peut retrouver les nuances manquantes dans les couleurs les plus utilisées dans l'image; cependant il se peut que certaines couleurs bien présentes soient éliminées par la même occasion. Avec l'exemple précédent (le champs de pissenlits avec le coquelicot solitaire), les nuances dans le ciel et dans les jaunes seront certainement bien rendues, mais le coquelicot a de bonnes chances de virer au jaune, au bleu ou au vert. Pire encore : si le ciel contient tellement de nuances de bleu que chacune d'entre elles n'apparaît qu'un très petit nombre de fois, elles risquent de ne pas être retenues dans la table de couleur finale même si la quantité globale de bleu est très importante dans l'image.

La *median cut color quantization* combine le meilleur des deux approches précédentes afin de produire une palette qui, dans la mesure du possible, n'élimine aucune couleur complètement, et retient en nombre suffisant les couleurs qui nécessitent de nombreuses nuances dans l'image.

Afin de voir comment fonctionne cette technique, commencez par imaginer l'espace des couleurs RVB comme un cube tri-dimensionnel (Figure ???).

Les axes du cube correspondent aux trois couleurs primaires : rouge, vert et bleu. Comme un point dans un graphe 3D, une couleur RVB sera représentée par un point dans le cube en prenant comme coordonnées le long des trois axes les valeurs de ses composantes R, V et B. Puisque chaque composante est codée sur 8 bits, chaque axe du cube des couleurs va de 0 à 255. Si on dessinait dans le cube 3D chaque couleur présente dans l'image, on s'apercevrait que le cube n'est pas entièrement rempli, certaines régions contenant un grand nombre de points alors que d'autres en contiennent très peu ou pas du tout.

L'idée derrière la *median cut color quantization* est que les régions du cube les plus denses vont produire proportionnellement plus de couleurs dans la table de couleur que les zones peu denses. Ainsi on "biaise" la table de couleurs afin que les couleurs dans l'image qui nécessitent de nombreuses nuances (le bleu dans le ciel) seront rendues à partir d'un plus grand nombre de couleurs dans la table que celles qui sont peu fréquentes (le coquelicot rouge).

Choisir les couleurs de cette manière est plus simple qu'il n'y paraît au premier abord. Le mécanisme brillant derrière cette méthode est une simple procédure qui va diviser récursivement la table des couleurs afin de refléter la distribution réelle des couleurs dans l'image.

L'algorithme est le suivant :

- Tout d'abord, une boîte imaginaire est dessinée au plus près autours des points du cube qui représentent les couleurs dans l'image. Cette boîte n'est pas forcément de la taille du cube. Elle peut être plus petite puisqu'elle "serre" les points au plus près. Si aucune couleur n'a de composante rouge inférieure ou égale à 8 ($R <= 8$), un des côtés de la boîte sera exactement aligné sur le plan perpendiculaire à l'axe R, à la valeur 8.
- Ensuite, l'algorithme prend le côté le plus long de la boîte et le divise en deux au point médian. Ce point n'est pas nécessairement égal au point milieu. S'il y a un million de pixels dans l'image et si la boîte est en train d'être divisée le long de l'axe R, on va trier les un million de valeurs des composantes R des pixels par ordre croissant, et le point médian sera celui correspondant à la 500.000 valeur. S'il y a beaucoup plus de points avec une composante R faible que de points ayant une composante R forte, alors le point médian peut très bien tomber sur la valeur $R = 60$, ce qui n'est pas le milieu de l'axe (qui est à 255 div 2).
- La boîte est maintenant divisée en deux boîtes plus petites, chacune contenant à peu près une

"constitution" égale de couleurs.

- La même procédure est maintenant appliquée aux deux boîtes : à chacune on associe une boîte qui serre les couleurs au plus près, on choisit le plus grand côté et on coupe en deux. Comptons les boîtes : deux boîtes qu'on coupe encore en deux, ça nous fait donc quatre boîtes maintenant. Elles n'ont pas la même taille mais contiennent chacune à peu près le même nombre de couleurs.
- On répète ce processus jusqu'à ce qu'on ait un nombre de boîtes égal à 256 (ou au nombre de couleurs affichables simultanément sur la machine dont on dispose).
- Une fois que les subdivisions sont effectuées, on peut affecter nos 256 couleurs en prenant la valeur RVB qui se trouve au centre de chaque boîte. Si il n'y a rien au centre de la boîte, on calcule la valeur RVB correspondant à la moyenne des valeurs RVB des pixels à l'intérieur de la boîte.

Le résultat de cet algorithme est une table de couleur qui contient de manière optimale un échantillonnage des couleurs présentes dans l'image. Les nombreuses nuances de bleu de notre exemple résultent en une forte concentration de teintes bleues dans la table de couleur. Et le rouge du coquelicot est bien présent car au moins une des boîtes a contenu les quelques nuances de rouge qui le caractérisent.

Les programmes de traitement d'image comme Adobe Photoshop ou XV utilisent cette technique. En général l'option proposée est *"use adaptative palette"* ou *"choose best colormap"*.

3 Tramage de Floyd-Steinberg

Une table de couleur bien choisie est un grand pas dans la bonne direction, mais ce n'est pas encore suffisant! Un système 256 couleurs ne peut afficher que 256 couleurs simultanément, pas une de plus! Pour obtenir un rendu de qualité il manque une chose : le tramage!

Le tramage arrange les pixels de telle manière que les couleurs "se mélangent" et donnent l'impression que l'image contient plus de couleurs que les 256 dont on dispose dans la table de couleurs. Imaginez que la table contienne seulement deux couleurs -bleu et jaune- et que l'on veuille afficher du vert. En dessinant alternativement un pixel bleu, un pixel jaune, un pixel bleu, etc.... on arrive a créer un bloc de couleurs qui ressemble à du vert. C'est cela le tramage, dans sa forme la plus simple.

Il est important d'avoir un bon algorithme de tramage, capable de traiter des images quelconques, ne contenant pas forcément de blocs de couleur uniforme. Le tramage de Floyd-Steinberg est parmi les meilleurs algorithmes disponibles. Un programme qui utilise l'approche de Floyd-Steinberg examine chaque pixel dans l'image, change sa couleur pour la plus proche dans la table de couleurs et calcule une erreur égale à la différence entre la valeur RVB de la couleur approchée et la valeur RVB de la couleur réelle. Ensuite, il distribue, il *diffuse* cette erreur sur les pixels voisins. Lorsque on applique ce traitement à tous les pixels pris un par un, le résultat est une image extrêmement fidèle à l'originale, mais qui n'utilise que les couleurs de la table.

La figure ??? montre comment l'erreur sur la couleur est calculée pour un pixel et distribuée à ses voisins. Supposons que la valeur actuelle d'un pixel est R=100, V=120 et B=140, et que la couleur la plus proche dans la table de couleurs est R=128, G=128, B=128. Pour chaque composante l'erreur est ainsi égale à R=-28, V=-8, B=12. Cette erreur est distribuée entre le pixel à droite du pixel de départ et les trois pixels au-dessous en utilisant le filtre de Floyd-Steinberg, décrit lui aussi figure ???. Le pixel immédiatement à droite récupère 7/16 de l'erreur. Sa valeur R est diminuée de 7/16$^{\text{ème}}$ de 28 (à peu près 12), sa composante V est diminuée de 7/16ème de 8 (environs 3), sa composante B est augmentée de 7/16ème de 18 (environs 8). SI la valeur originelle de ce pixel est R=160, V=168, B=140, alors il prend la valeur R=148, V=165, B=148.

Le pixel au-dessous à droite du pixel cible est modifié de la même manière, mais il ne récupère que 1/16ème de l'erreur totale sur la couleur. Le pixel en-dessous du pixel cible récupère 5/16ème de l'erreur, celui en-dessous à gauche 3/16ème.

La somme de 1/16, 3/16, 5/16 et 7/16 est égale à 1, ainsi toute l'erreur sur la couleur du pixel cible a été distribuée. Globalement, aucune erreur supplémentaire n'a été introduite dans l'image.

Ce processus est appliqué une seule fois à tous les pixels de l'image. La forme du filtre, combinée avec l'ordre dans lequel les pixels sont traités, assure qu'aucune erreur n'est distribuée à un pixel après qu'une couleur lui ait été attribuée dans la table.

Un programme de tramage qui utilise le filtre de la figure ??? commence par traiter le pixel en haut à droite de l'image et avance horizontalement de gauche à droite jusqu'à ce qu'il atteigne la fin de la ligne. Il passe ensuite à la ligne suivante et recommence jusqu'à ce que toutes les lignes aient été traitées.

Il existe de nombreuses versions de ce tramage, adaptées à des besoins particuliers, qui utilisent d'autres valeurs pour le filtre, qui traitent les pixels dans un ordre différent, etc...

La figure ??? montre les différents effets de l'optimisation de palette et du tramage, seuls et en combinaison.

A TERMINER !!!!!!!

4 Tramage ordonné (ordered dithering)

4.1. Introduction

En synthèse d'image, lorsqu'on dessine une pixel (x, y), on ne connaît pas forcément la couleurs des pixels voisins, et l'algorithme de Floyd Steinberg ne peut plus s'appliquer.

Rappel : pour étendre la palette des intensités visibles sur un écran, il faut utiliser une technique de tramage, basée sur le fait que nos yeux procèdent à une intégration spatiale des données observées. Si l'on regarde une zone particulière de l'écran à une certaine distance, nos yeux ne voient plus les pixels individuellement, mais filtrent leurs intensités et ne captent que l'intensité moyenne émise par les pixels de la zone formant un cercle autour du point visé.

Ce phénomène est exploité par les imprimeurs : dans les quotidiens les photographies sont tramées, les couleurs approximées à l'aide de motifs bicolores n'utilisant que le noir et le blanc. Ces motifs sont composées de cercles noirs plus ou moins importants. Leur taille varie fonction de l'intensité désirée dans telle ou telle zone de l'image.

Sur les écrans d'ordinateurs, il est impossible d'utiliser exactement la même technique, mais le principe reste similaire. Pour approximer une zone d'une certaine intensité, on utilise des matrices de tramage. Par exemple, à l'aide d'une matrice $2x2$, si l'on ne dispose que de deux couleurs, le noir et le blanc, on pourra approximer plusieurs niveaux de gris selon qu'on allume ou qu'on éteint certains pixels.

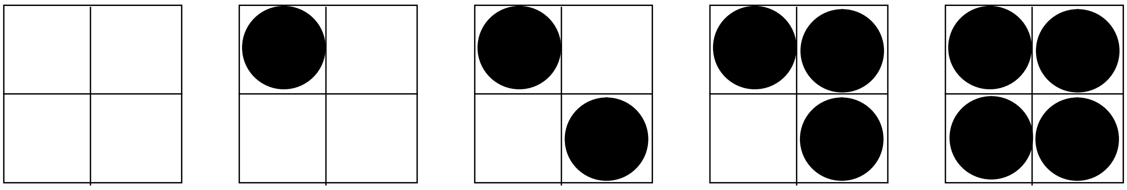


FIGURE 1. Une matrice de dispersion 2x2 approxime 5 intensités

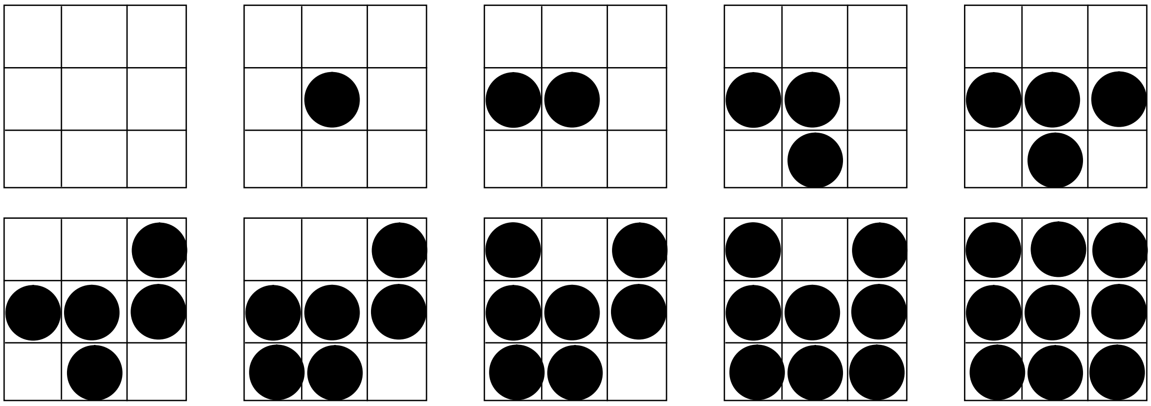


FIGURE 2. Les intensités consécutives obtenues avec une matrice 3x3

En pratique, avec une matrice $n \times n$, on peut approximer $n^2 + 1$ intensités, si l'on ne dispose que de deux valeurs pour chaque pixel.

Il faut cependant faire attention à ne pas faire grossir démesurément la taille de la matrice de tramage car l'œil humain possède une acuité limitée. Il faut donc, lorsqu'on construit un modèle de tramage, tenir compte de ce facteur, de la distance à laquelle les spectateurs vont regarder, de la résolution de notre écran (taille réelle des pixels), etc... Il va de soit que si on construit une mosaïque de 100 mètres carrés, situées sur un des côtés du centre Pompidou, on pourra utiliser une plus grande matrice de tramage.

Si l'image est plus grande que la taille de la matrice de tramage, il suffira de prendre les coordonnées d'un point(x, y) modulo n.

4.2. Codage des matrices de tramage

4.2.1. Principe général

En pratique, comment coder ces matrices ? La solution la plus simple consiste à stocker dans chaque case de la matrice une intensité.

Pour dessiner une zone d'intensité I, on allumera toutes les cases dont la valeur est inférieure à I.

Par exemple, avec une matrice :

$$M = \begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix} \qquad \text{(EQ 1)}$$

...on peut approximer 10 intensités, comme le montre la Figure 5 page 8.

Les valeurs de la matrice ne sont pas attribuées au hasard. Leur distribution doit vérifier certaines contraintes :

- Ne pas créer de motifs trop identifiables. Par exemple, le motif de la Figure 3 page 6 est à éviter absolument. Il est évident que si ce motif se répétait, il provoquerait l'apparition de lignes horizontales dans l'image, qui seraient du plus mauvais effet.

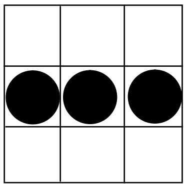


FIGURE 3. Motif à éviter car trop facilement discernable

- Les motifs doivent former une séquence cohérente : si un pixel est allumé pour une intensité i, il doit aussi demeurer allumé pour une intensité $k > j$. Ceci minimise les différences entre motifs d'intensité voisine et évite de faire apparaître des frontières trop visibles dans les dégradés de couleurs.
- Les motifs doivent s'étendre à partir du centre de la matrice, pour donner l'impression d'un disque qui grossit (se rappeler le fonctionnement de la vision humaine). Cette contrainte se traduit par le fait que des pixels allumés pour une intensité donnée doivent êtres voisins dans la matrice. L'exemple ci-dessous est par conséquent inacceptable :

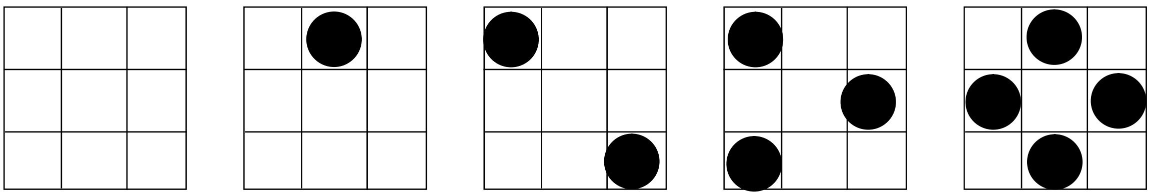


FIGURE 4. Motifs inacceptables car ne grossissant pas logiquement

4.2.2. Calcul des matrices de tramage

De nombreuses méthodes ont été proposées pour calculer des matrices de tramage répondant à tous ces critères. Holladay [HOLL80] a proposé une méthode qui a longtemps fait référence, surtout pour des sorties papier bicolores.

Pour le tramage sur écran, Bayer [BAYE73] a proposé des matrices de tramage minimisant la perceptibilité des motifs introduits.

Suivant sa méthode, une matrice de dispersion de taille n est notée $D^{(n)}$. Pour la calculer on part d'une matrice 2×2 semblable à celle-ci :

$$D^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \qquad \text{(EQ 2)}$$

...qui a été utilisé pour calculer l'illustration de la Figure 1 page 5.

Pour calculer de plus larges matrices, on utilise une relation de récurrence permettant de passer de $D^{(n)}$ à $D^{(2n)}$.

Soit $U^{(n)}$ une matrice $n \times n$ ne comportant que des 1 dans toutes ses cases. La relation de récurrence peut s'exprimer ainsi :

$$D^{(n)} = \begin{bmatrix} 4D^{(n/2)} + D_{00}^{(2)}U^{(n/2)} & 4D^{(n/2)} + D_{01}^{(2)}U^{(n/2)} \\ 4D^{(n/2)} + D_{10}^{(2)}U^{(n/2)} & 4D^{(n/2)} + D_{11}^{(2)}U^{(n/2)} \end{bmatrix} \qquad \text{(EQ 3)}$$

En appliquant cette relation sur $D^{(2)}$ pour calculer $D^{(4)}$, on obtient :

$$D^{(4)} = \begin{bmatrix} \begin{bmatrix} 0 & 8 \\ 12 & 4 \end{bmatrix} & \begin{bmatrix} 2 & 10 \\ 14 & 6 \end{bmatrix} \\ \begin{bmatrix} 3 & 11 \\ 15 & 7 \end{bmatrix} & \begin{bmatrix} 1 & 9 \\ 13 & 5 \end{bmatrix} \end{bmatrix} \qquad \text{(EQ 4)}$$

... et ainsi de suite...

4.3. Tramage non monochrome

Le tramage présenté dans les figures précédente suppose qu'on utilise un écran monochrome, cependant la méthode s'applique aussi si l'on dispose d'un écran possédant des caractéristiques différentes. Par exemple, avec 2 bits par pixels, on peut afficher 4 intensités différentes sur un pixel. La technique de tramage étudiée dans le cas monochrome est applicable et les plus grandes variations d'intensités sur un pixel vont permettre d'accroître grandement le nombre d'intensités affichables.

Exemple : une matrice de taille $2x2$, correspondant à des motifs de 4 pixels, chacun pouvant prendre, en plus du noir, 3 autres valeurs, permettra d'afficher $I_{total} = (2^2 \times 3) + 1 = 13$ intensités différentes :

4.4. Algorithme d'adressage, cas non monochrome

L'adressage est légèrement plus compliqué que dans le cas monochrome. Examinons la Figure 5 page 8. Remarquons dans un premier temps que la somme des intensités des pixels situées dans les matrices 0..12 est égale à l'intensité approximée.

On remarque ensuite que pour chacune des 13 matrices, seules deux valeurs apparaissent : 0 et 1, 1 et 2, 2 et 3. Jamais 0 et 3. C'est normal, car on utilise une méthode dérivée de celle s'appliquant au cas monochrome, seules deux intensités sont utilisées pour former un motif approximant une intensité non disponible.

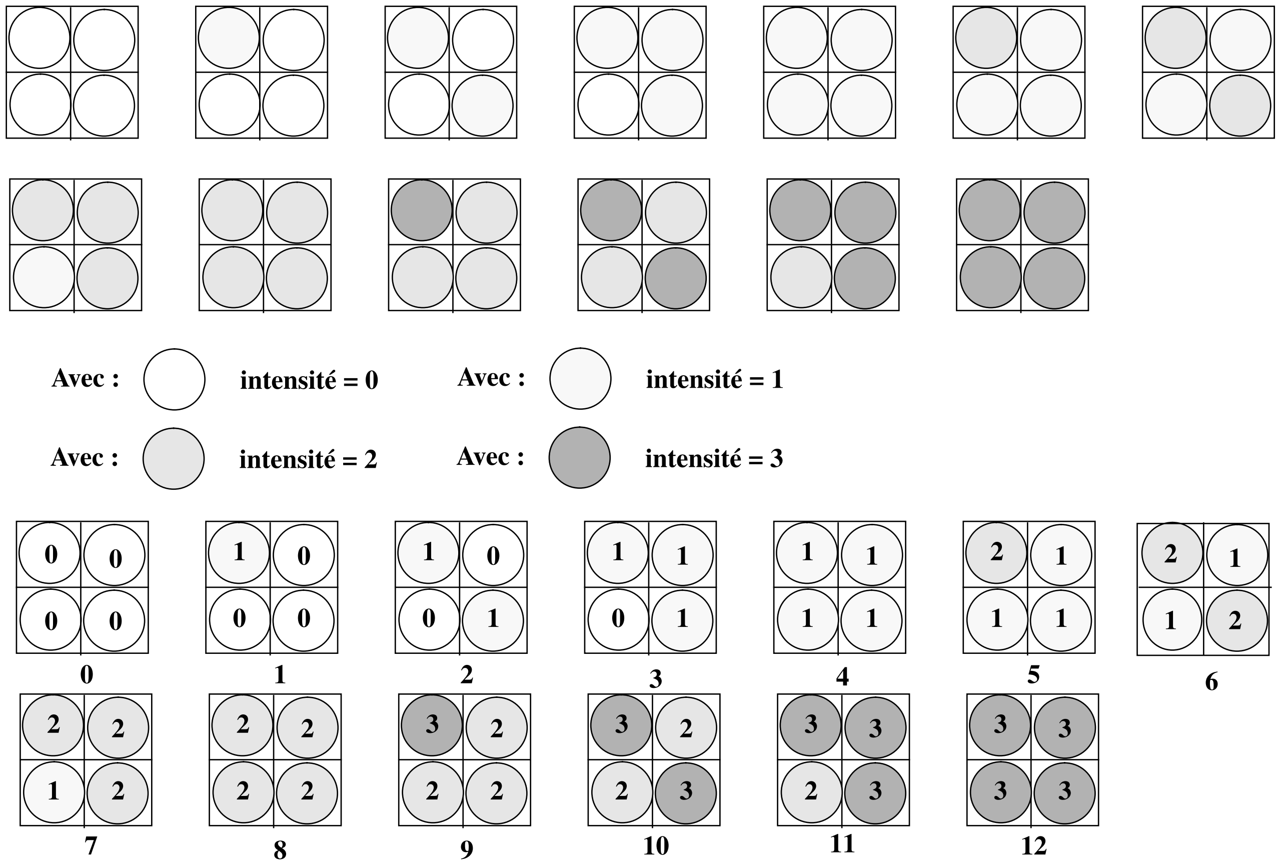


FIGURE 5. a 13 intensités peuvent être approximées par une matrice 2x2 si on dispose de 4 valeurs par pixel

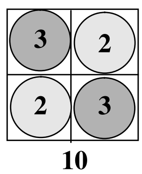


FIGURE 5. b La somme des valeurs est égale à l'intensité désirée

Prenons l'exemple de la Figure 5 page 8 : pour approcher l'intensité I=10, on a une matrice dont la somme des cases est égale à $3 + 2 + 3 + 2 = 10$!

4.5. Méthode d'adressage dans le cas non monochrome

Soit $D^{(n)}$ une matrice de tramage de taille $n \times n$ et p le nombre de bits par pixels que la carte graphique de notre ordinateur est capable de gérer. Le nombre d'intensités possibles pour un pixel est égal à 2^p, cependant, grâce au tramage, le nombre maximal d'intensités que l'on peut approcher par des motifs est égal à $I_{max} = n^2 \times (2^p - 1) + 1$.

On désire dessiner avec l'intensité I_d comprise entre 0 et $I_{max} - 1$. Comment choisir le motif permettant d'approximer cette intensité ? Comment choisir la couleur de chacun des pixels du motif ?

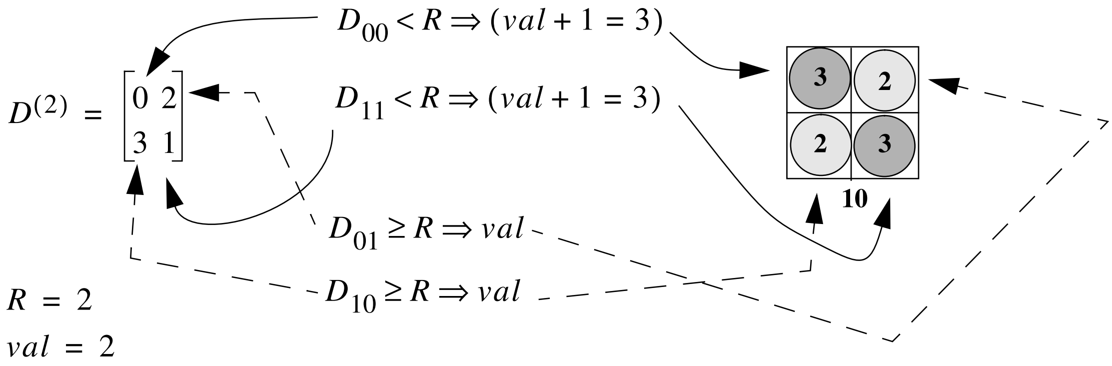
L'algorithme se décompose en plusieurs étapes.

1. Calculer les deux intensités (parmi celles disponibles) des pixels qui vont composer le motif.
Si on dispose de p bits par pixels, toutes les intensités des pixels seront strictement inférieures à 2^p donc il suffit de calculer $val = I_d / 2^p$. Le motif ne contiendra que des pixels d'intensités val et $val + 1$.

2. Déterminer l'intensité d'un pixel $P(x, y)$ appartenant à une image.
On doit choisir entre val et $val + 1$. On calcule tout d'abord $R = l_d$ modulo 2^p le reste de la division de l_d par 2^p . Les motifs se répétant dans l'image, on calcule ensuite $u = x$ modulo n et $v = y$ modulo n afin de s'assurer que l'on puisse utiliser des coordonnées se trouvant dans les limites de la matrice de tramage. On compare ensuite R avec la case $D_{uv}^{(n)}$ de la matrice de tramage. Si $D_{uv}^{(n)} < R$ alors le pixel $P(x, y)$ sera d'intensité $val + 1$ sinon d'intensité val .

Revenons sur l'exemple. On désire approximer une intensité $I_d = 10$.

- On calcule $val = 10/4 = 2$ (division entière)
- On calcule $R = 10$ modulo $4 = 2$



5.0 Mise en pratique

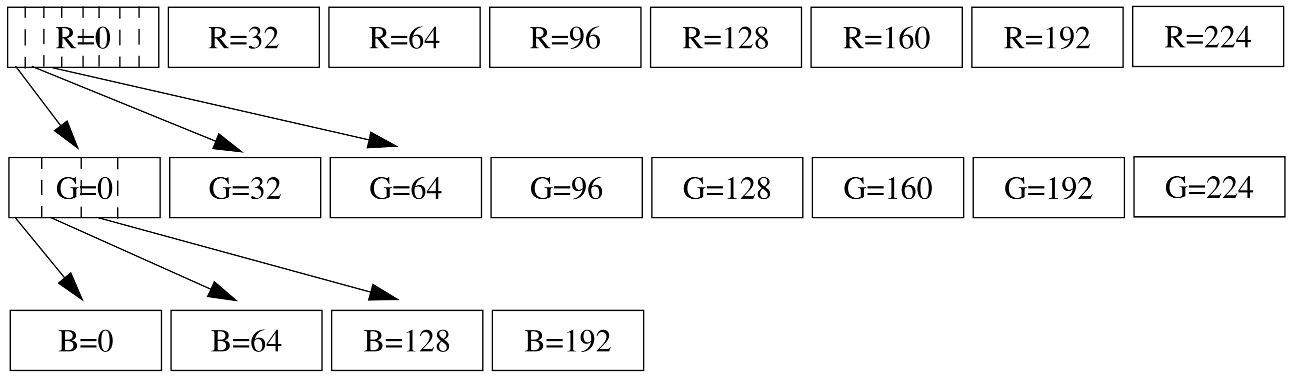
5.1. Construction de la table de couleurs, fonction d'adressage rapide

Dans notre programme de TD, la table de couleur est allouée de la manière suivante :

```
/*---------------------------------------*/
void Lut_16M(Display *dpy, Colormap cmap)
/*---------------------------------------*/
{
  long r,g,b,a=0;

  for (r=0;r<8;r++)
    for (g=0;g<8;g++)
      for (b=0;b<4;b++)
        SetRGB_CMap(dpy, cmap ,a++ , r<<5, g<<5, b<<6);
}
```

Soit une palette composée de 8 niveaux de rouge, 8 niveaux de vert et 4 niveaux de bleu. En fait, l'adressage se fait de la manière suivante :



Dans la mémoire, cette table de couleur est contigue : on retrouve d'abord toutes les couleurs ayant un niveau de rouge égal à zéro, ensuite la zone mémoire contenant les couleurs de niveau de rouge égal à 1, etc...

Revenons un instant sur l'instruction suivante :

```
SetRGB_CMap(dpy, cmap ,a++ , r<<5, g<<5, b<<6);
```

Pour quoi faire des décalages ici ? En fait le décalage r << 5 revient à divisier le nombre de cases disponibles dans la table de couleurs (255) par le nombre de niveaux que l'on désire pour la composante rouge (8). Si on divise 255 par 8 on obtient $2^8/2^3 = 2^5$, que l'on peut aussi coder en utilisant un décalage de 5 bits à gauche. Il y a donc dans la table de couleurs un écart de 2^5 entre les différents niveaux de rouge. Idem pour le vert. Pour le bleu, comme on a que quatre niveaux, ils varient de 0 à 192 par pas de 64.

En fait, c'est comme si on avait écrit :

```
nb_cols_dans_la_table = 255;
nb_niveaux_rouge = 8;
nb_niveaux_vert  = 8;
nb_niveaux_bleu  = 4;
```

```
increment_rouge = nb_cols_dans_la_table/nb_niveaux_rouges;
increment_vert  = nb_cols_dans_la_table/nb_niveaux_vert;
increment_bleu  = nb_cols_dans_la_table/nb_niveaux_bleu;

for(i = 0; i < nb_niveaux_rouge; i++)
  rouge = i * increment_rouge;
  for(j=0; j < nb_niveaux_bleu; j++)
     vert = j * increment_vert;
     for(k = 0; k < nb_niveaux_bleu; k++)
        bleu = k * increment_bleu

SetRGBColormap(dpy, cmap, a++, rouge, vert, bleu)
```

Entre zones mémoires correspondant à un niveau de rouge constant il y a 32 cases (chaque case de rouge contient 8 cases de vert qui elles-même contiennent 4 cases de bleu). Dans chacune de ces zones on distingue 8 zones contiguës correspondant à un niveau de vert, séparées entre elles de 4 cases correspondant aux niveaux de bleu.

Cette construction systématique permet l'utilisation d'une macro permettant un adressage très rapide :

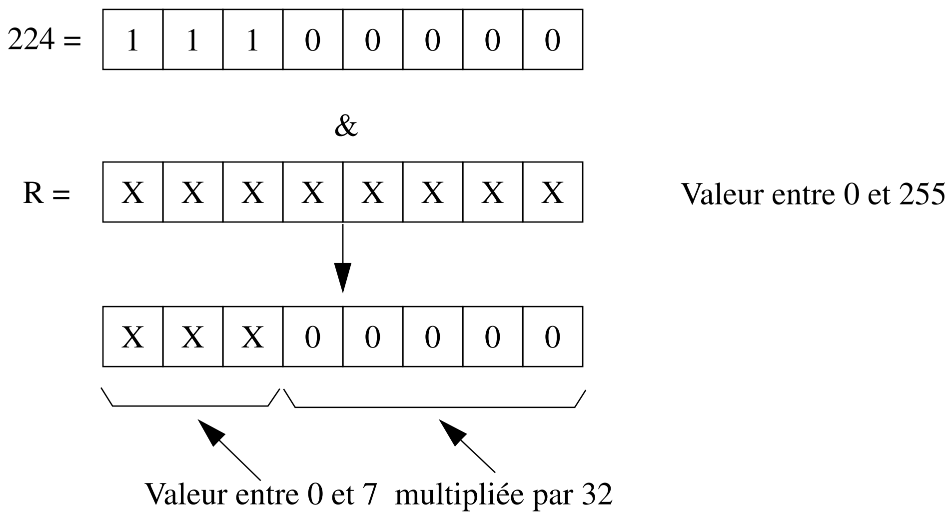
```
#define RGBTOONE(r,g,b) (((r) & 224) + (((g) & 224)>>3) + (((b) & 192)>>6))
```

Nous allons voir comment fonctionne cette macro. Tout d'abords, les valeurs R, G et B passées en paramètres sont comprises entre 0 et 255. La macro retourne l'index de la case dans la colormap contenant la couleur la plus proche des valeurs R, G et B demandées.

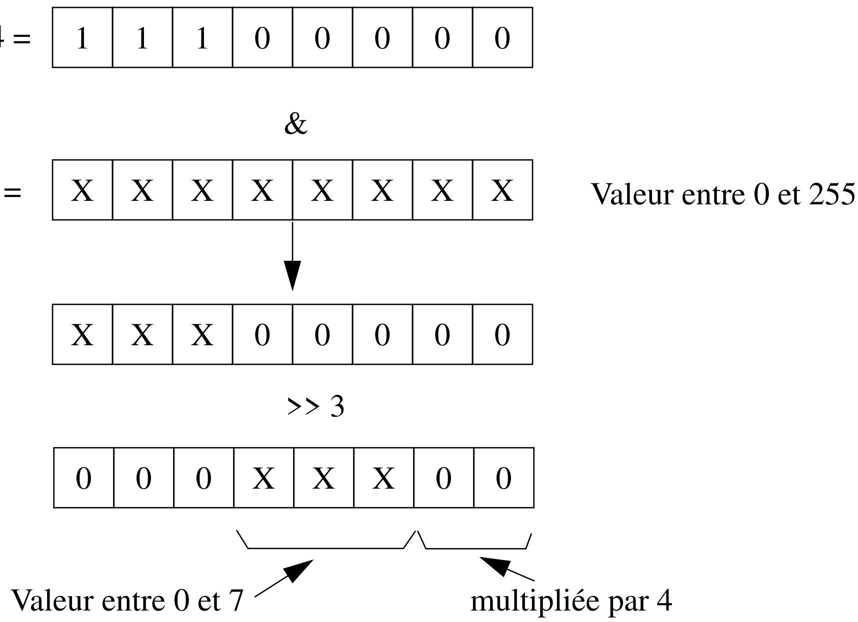
Les trois parties de l'addition correspondent aux décalages mémoire entre les différentes zones qui composent la colormap.

- L'instruction (r & 224) ne conserve du rouge que les 3 bits les plus à gauche, rappelez-vous, on avait décalé l'intensité de rouge de 5 bits à gauche dans la fonction Lut16M(), les intensités de rouges ne sont donc codées que dans les 3 bits les plus à gauche de l'octet. En masquant par la valeur 224, on ne garde que le modulo 32 de la valeur R. C'est-à-dire le numéro de la case dans la colormap ou va commencer la zone d'intensité constante R. Par exemple, pour R=0, ca donne 0, pour R=33, c'est-a-dire 0010 0001, en masquant par 1110 0000, on obtient 0010 0000 = 32, etc.... La case correspondant à R=33 G=0 B=0 se trouve bien à la case 32. Ce résultat est vrai pour les intensités ayant R entre 32 et 64 non inclu. En fait on obtient un résultat n entre 0 et 7 (puisque'on a 8 niveaux de rouge), que l'on multiplie par 32 (les cinq zéros à droite), ca nous donne bien l'adresse du début de la zone R=n
- L'instruction ((g & 224) >> 3) est presque similaire : on ne conserve de la variable G que les 3 bits les plus à gauche car les intensités varient de 32 en 32. En masquant, passe de l'intervalle de valeur [0, 255] à [0, 7], puisqu'on a 8 niveaux de vert.

  Mais pourquoi décaler de 3 bits à droite (ce qui revient à diviser la valeur obtenue par $2^3 = 8$) ? Supposons que l'on cherche la case correspondant à l'intensité R=33 G=33 B=0. On sait, d'après ce que l'on vient de voir pour le rouge, que la case correspondante se trouve quelque part entre la position 32 et la position 64 (zone correspondant à des valeurs de R comprises entre 32 et 64), il nous faut maintenant calculer à partir de la valeur de G le nombre de cases à sauter à partir de la position 32. Avec le masquage par 224 on a G entre 0 et 7. Les valeurs successives de G sont espacées de 4 cases,



car dans chaque zone mémoire où G est constant se trouvent 4 valeurs de bleu. Il faut donc multiplier la valeur de G entre 0 et 7 par 4, ce qui correspond à la décaler de 3 bits vers la droite puisque elle se trouvait déjà dans les 3 bits les plus à gauche de l'octet, à cause du masquage par 224.



- L'instruction (((b) & 192)>>6) s'explique de la même manière. Le masquage transforme la valeur passée en paramètres, comprise entre 0 et 255, en une valeur comprise entre 0 et 3 (il n'y a que 4 niveaux de bleu), située sur les deux bits les plus à gauche de l'octet. Le nombre de cases à sauter dans une zone de vert constant est donc égale à la valeur représentée par ces deux bits. Il suffit de les décaler de 6 vers la droite pour que la valeur finale de l'octet soit égale à celle de ces deux bits.

5.2. Fonction de tramage

La fonction de tramage utilise globalement la méthode d'adressage que nous venons de voir, et la mixe avec les méthode de tramage étudiées dans la section 4 page 4. Rappelons que nous désirons nous approcher d'un modèle de couleurs où on disposerait de 255 niveaux de rouge, de vert et de bleu. Dans la table de couleur nous avons créé une palette mélangeant 8 niveaux de rouge, 8 niveaux de vert et 4 niveaux de bleu (l'œil humain voit moins bien le bleu que les autres nuances). Nous choisirons donc une matrice de tramage plus grande pour le bleu que pour le rouge et le vert.

Quelles tailles choisir pour ces matrices ?

Etudions tout d'abord le cas du rouge et du vert. La théorie (voir Figure 5 page 8) nous dit que si on dispose d'une matrice $n \times n$, et de p nuances par pixels en plus du noir, on peut approximer $n^2 \times (p-1) + 1$ intensités.

Pour les composantes rouge et verte, nous disposons de 7 niveaux d'intensité en plus du noir. Nous prendrons par conséquent n tel que $n^2 \times 7 + 1 \leq 255$, avec n puissance de deux (car on construit une matrice de tramage de taille $2n$ à partir d'une matrice de taille n). Avec $n = 4$ ça nous donne $4^2 \times 7 + 1 = 113$ intensités. Si on prend $n = 8$ on peut approximer 449 intensités différentes, ce qui est trop important pour ce que l'on veut faire.

Pour la composante bleu, on ne dispose que de 3 intensités en plus du noir. On prendra donc $n = 8$ ce qui permet d'approximer $8^2 \times 3 + 1 = 193$ intensités.

On reconnaît dans le code de la Figure 6 page 14 les deux matrices D4 et D8, la première consacrée au tramage des rouges et des verts, la seconde au tramage des bleus.

Quelques explications sont cependant nécessaire car ce code est assez peu lisible tel quel. Prenons l'exemple d'une couleur ne comportant que du rouge. Par exemple la valeur R=255 G=0 B=0.

- On transforme **`x`** et **`y`** dans l'intervalle [0..3] (en fait en faisant **`x&3`** et **`y&3`** on calcule leur modulo 3)
- On calcule **`r = R >> 1`** car on doit passer d'un intervalle de valeurs [0..255] à un intervalle [0..113], puisqu'avec le tramage choisi : une matrice 4×4 et 7 niveaux de rouge, on ne peut approximer que $I_{max} = 113$ intensités. En divisant R par deux on effectue une approximation du changement de repère (un changement exact aurait coûté trop cher en instructions machine).
- On calcule **`modr = r & 15`**. Ceci revient à calculer le reste de la division de **`r`** par 16. Rappelons que les valeurs situées dans la matrice de tramage sont toutes entre 0 et 15.
- **`r = r >> 4`**, avec le premier décalage de 1 à droite au début, cela revient à décaler R de 5, soit à le diviser par 32. En divisant une valeur comprise au départ entre 0 et 255, par 32, on obtient une valeur entre 0 et 7. Normal, on dispose de 8 niveaux de rouge, compris entre 0 et 7. Les intensités des pixels composant le motif seront donc égales à **`R >> 5`** ou a **`(R>>5) + 1`**. Pour décider...
- ...on compare **`modr`** aux cases de la matrice. Si la valeur de la case de la matrice est plus petite, alors le pixel a comme couleur **`(R>>5) +1`**, sinon **`(R >> 5)`**.
- ...raisonnement similaire avec le vert et le bleu...

```
unsigned char Dither(long x, long y, unsigned char R, unsigned char G,
                      unsigned char B)
/* Renvoie l'index dans une colormap preparee de la couleur 24 bits du
  pixel a la position x, y */
{
    static unsigned char D8[8][8] =
    {
        {0, 32, 8, 40, 2, 34, 10, 42},
        {48, 16, 56, 24, 50, 18, 58, 26},
        {12, 44, 4, 36, 14, 46, 6, 38},
        {60, 28, 52, 20, 62, 30, 54, 22},
        {3, 35, 11, 43, 1, 33, 9, 41},
        {51, 19, 59, 27, 49, 17, 57, 25},
        {15, 47, 7, 39, 13, 45, 5, 37},
        {63, 31, 55, 23, 61, 29, 53, 21}
    };
    static unsigned char D4[4][4] =
    {
        {0, 8, 2, 10},
        {12, 4, 14, 6},
        {3, 11, 1, 9},
        {15, 7, 13, 5}
    };
    short d4tmp = D4[x & 3][y & 3];
    short r, v, b, modr, modv, modb;

    r = R >> 1;
    v = G >> 1;
    b = B;

    modr = r & 15;
    modv = v & 15;
    modb = b & 63;

    r >>= 4;
    v >>= 4;
    b >>= 6;

    if ((modr >= d4tmp) && (r != 7))
        r++;
    if ((modv >= d4tmp) && (v != 7))
        v++;
    if ((modb >= D8[x & 7][y & 7]) && (b != 3))
        b++;
    return ((unsigned char) ((r << 5) + (v << 2) + b));
}
```

FIGURE 6. Fonction de tramage