Algorithmes de base

François Faure

Résumé

Au plus bas niveau, les images numériques sont représentées par des tableaux de valeurs communément appelées grilles (*raster graphics*). Les operations de base consistent à tracer des primitives géométriques simples et à remplir des polygones.

Table des matières

1 Grilles **2**

2 Algorithme de Bresenham **2**

3 Remplissage de polygones **6**
3.1 Cas du rectangle 6
3.2 Cas général 6
3.3 Traitement des couleurs 6

4 Tracés dans OpenGL **7**

MCours.com

1 Grilles

On définit une image comme un tableau de valeurs, chaque valeur représentant la couleur d'un certain point (*pixel*). L'aspect discret (échantillonné) de l'image s'atténue si on la regarde de suffisamment loin. On utilise différents termes pour représenter de telles grilles, suivant le type des valeurs contenues :

- **bitmap** les valeurs sont binaires et représentent par exemple le noir et le blanc ;
- **greymap** les valeurs sont plus nuancées et représentent par exemple des niveaux de gris ;
- **pixmap** les valeurs comportent plusieurs composantes (canaux) afin de représenter des couleurs.

On crée une image en affectant une valeur à chacune des cases de la grille. Ces valeurs peuvent provenir directement d'un appareil à numériser (*scanner*) ou être calculées par un programme. Une image complexe se décompose généralement en *primitives graphiques* (segments de droites, arcs de cercle ou d'ellipses, polygones vides ou remplis, bitmaps, ...) qui sont tracées dans un certain ordre. Si un point fait partie de plusieurs primitives, sa valeur finale est le résultat d'une combinaison des valeurs associées à chacune d'elles. Ces combinaisons peuvent être plus ou moins complexes (opérateurs logiques, combinaisons linéaires, valeur maxi, dernière valeur spécifiée...) et le résultat peut dépendre de l'ordre dans lequel les primitives ont été tracées.

2 Algorithme de Bresenham

L'algorithme de Bresenham permet de tracer des lignes. Étant donné un segment de droite allant d'un point de départ x_1, y_1 (entiers) à un point d'arrivée x_2, y_2 (entiers) dans la grille, la question est : quels pixels noircir ? En effet, un échantillonnage du segment fournit en général des valeurs non entières. Il faut une méthode de choix des pixels qui garantisse la continuité du segment, la plus grande rectitude possible ainsi qu'un temps de calcul faible. La figure 1 représente un tel segment.

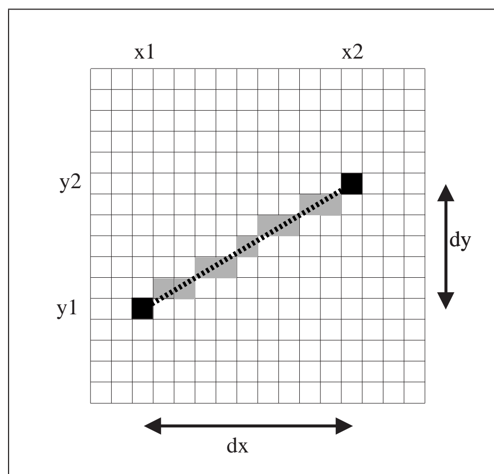

Fig. 1 – Un segment de droite échantillonné

Si la pente de la droite est inférieure à 1, alors nous devons allumer un et un seul pixel par colonne entre x_1 et x_2. Notez que ce n'est pas le cas pour les lignes. Nous pourrions donc écrire le programme suivant :

Calcul par l'équation de droite $y = mx + b$

```
dy = y2-y1;
dx = x2-x1;
m = dy/dx;
b = y1-m*x1;
for (x=x1; x<=x2; x++) {
  y=m*x+b;
  plot(x,round(y));
}
```

Voyons maintenant les améliorations successives aboutissant à un programme optimisé. En particulier, on désire éviter les calculs en virgule flottante pour ne traiter que des entiers.

Calcul de y par incrément

```
dy = y2-y1;
dx = x2-x1;
m = dy/dx;
y = y1;                                 /* <------------ */
for (x=x1; x<=x2; x++) {
  plot(x,round(y));
  y=y+m;                                /* <------------ */
}
```

Simplification de l'arrondi

```
dy = y2-y1;
dx = x2-x1;
m = dy/dx;
y = y1;
f = 0;                                /* <------------ */
for (x=x1; x<=x2; x++) {
  plot(x,y);
  f=f+m;                              /* <------------ */
  if (f>0.5) {                        /* <------------ */
    y = y+1;
    f = f-1;                          /* <------------ */
  }
}
```

Suppression de la division par dx

```
dy = y2-y1;
dx = x2-x1;
y = y1;
f = 0;
```

```
for (x=x1; x<=x2; x++) {
  f=f+dy;                                /* <------------ */
  plot(x,y);
  if (f>(dx/2)) {                        /* <------------ */
    y = y+1;
    f = f-dx;                            /* <------------ */
  }
}
```

Suppression de la division par 2

```
dy = y2-y1;
dx = x2-x1;
y = y1;
f = 0;
for (x=x1; x<=x2; x++) {
  plot(x,y);
  f=f+2*dy;                        /* <------------ */
  if (f>dx) {                      /* <------------ */
    y = y+1;
    f = f-2*dx;                    /* <------------ */
  }
}
```

Tester le signe au lieu de comparer

```
dy = y2-y1;
dx = x2-x1;
y = y1;
f = -dx;                               /* <------------ */
for (x=x1; x<=x2; x++){
  plot(x,y);
  f=f+2*dy;
  if (f>0)                             /* <------------ */
    y = y+1;
    f = f-2*dx;
  }
}
```

Ne modifier f qu'une seule fois par passage dans la boucle

```
dy = y2-y1;
dx = x2-x1;
y = y1;
f = -dx;
for (x=x1; x<=x2; x++) {
  plot(x,y);
  if (f>0) {
    y = y+1;
    f = f + 2*(dy-dx);                   /* <------------ */
```

```
  }
  else
    f = f + 2*dy;
}
```

Précalculer des valeurs

```
dy = y2-y1;
dx = x2-x1;
y = y1;
f = -dx;
incrE = 2*dy;                          /* <------------ */
incrNE = 2*(dy-dx);                    /* <------------ */
for (x=x1; x<=x2; x++) {
  plot(x,y);
  if (f>0) {
    y = y+1;
    f = f + incrNE;                    /* <------------ */
  }
  else
    f = f + incrE;                     /* <------------ */
}
```

Nous arrivons enfin à l'algorithme dit de Bresenham :

Sortir les bornes de la boucle

```
dy = y2-y1;
dx = x2-x1;
y = y1;
f = 2*dy-dx;
incrE = 2*dy;
incrNE = 2*(dy-dx);

plot(x1,y1);                       /* <------------ */
for (x=x1+1; x<x2; x++) {          /* <------------ */
  if (f>0) {
    y = y+1;
    f = f + incrNE;
  }
  else
    f = f + incrE;
  plot(x,y);
}
plot(x2,y2);                        /* <------------ */
```

Ce programme fonctionne pour les cas où $x_1 < x_2$ et $dy \leq dx$. Il est aisément adaptable aux autres cas en changeant des signes ou en permutant les rôles de x et y. Des algorithmes similaires existent pour tracer des arcs d'ellipses.

3 Remplissage de polygones

3.1 Cas du rectangle

On définit un rectangle par deux points diagonaux opposés x_1, y_1 et x_2, y_2. L'algorithme de remplissage consiste simplement en deux boucles imbriquées :

```
for( x=x1; x<=x2; ++x )
    for( y=y1; y<=y2; ++y )
        plot(x,y);
```

3.2 Cas général

Un polygone est défini par une liste ordonnée de points, chaque paire de points successifs définissant une arête. On emploie un algorithme basé sur une ligne de balayage (*scan-line*) qui parcourt un rectangle englobant le polygone ligne par ligne, comme illustré sur la figure 2. On parcourt chaque ligne en partant de l'extérieur et en comptant le nombre d'arêtes intersectées. Quand le nombre est impair on est à l'intérieur du polygone et on affiche la couleur désirée. Quand le nombre est pair on est à l'extérieur et on n'affiche rien. Les frontières sont déterminées d'une manière similaire à l'algorithme de Bresenham.

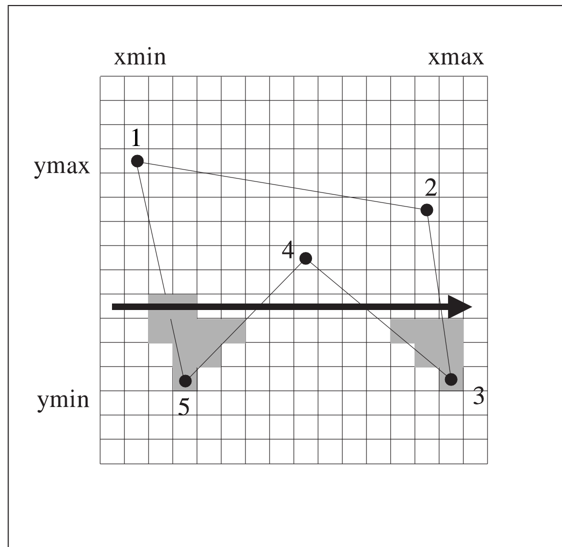

Fig. 2 – Balayage ligne par ligne d'un polygone.

3.3 Traitement des couleurs

La couleur d'un polygone peut étre uniforme ou interpolée (*Gouraud shading*). L'interpolation s'effectue d'abord linéairement sur les arêtes entre les sommets concernés. À l'intérieur, les couleurs sont interpolées entre les arêtes coupées par la ligne de balayage. La figure 3 présente des exemples de couleurs uniformes et interpolées.

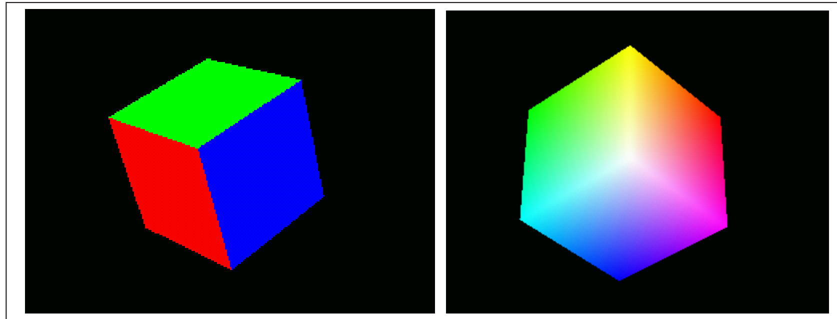

Fig. 3 – Coloriage de polygones. À gauche, couleurs uniformes. À droite, couleurs interpolées

4 Tracés dans OpenGL

En OpenGL, on peut utiliser des points comme primitives fondamentales, en spécifiant un des modes de tracé présentés sur la figure 4.

Le mode de coloriage est spécifié par glShadeModel(GL_FLAT) pour la couleur uniforme et glShadeModel(GL_SMOOTH) pour la couleur interpolée (mode activé par défaut).

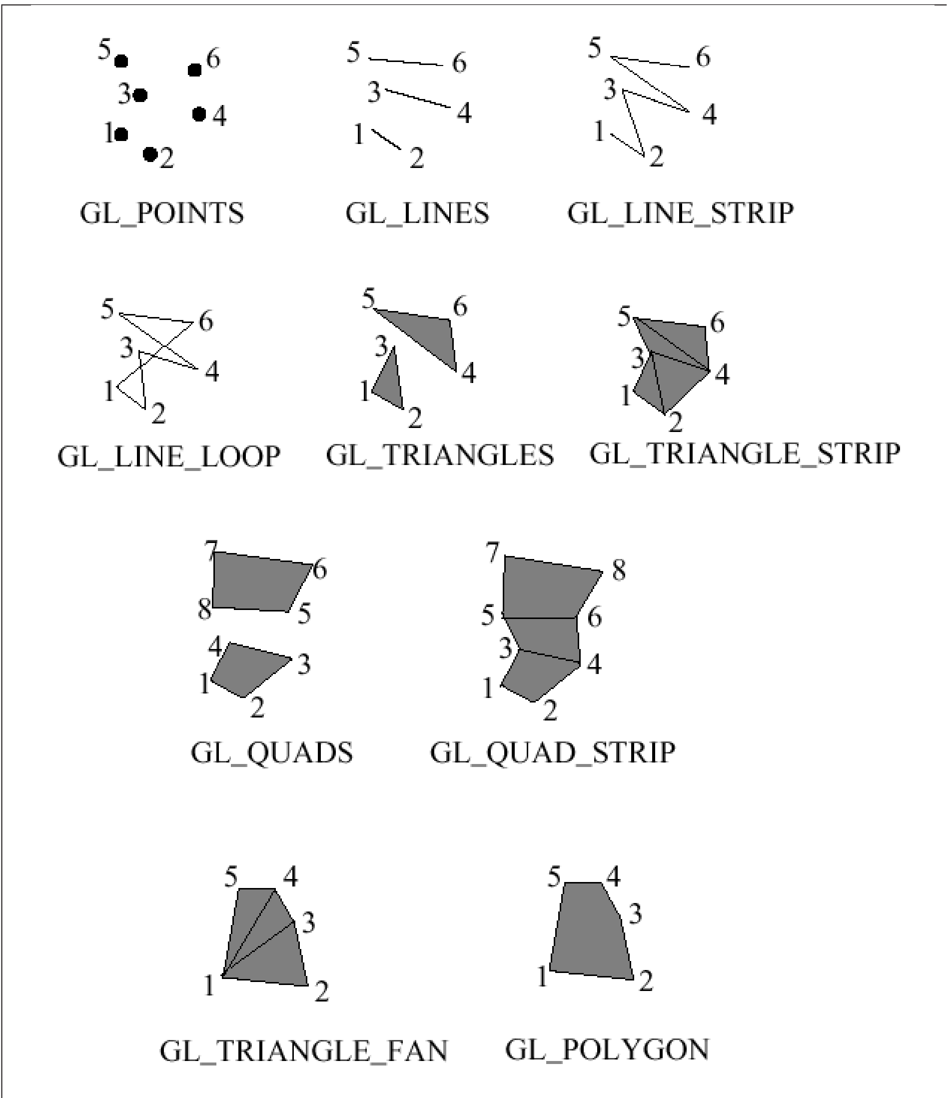

Fig. 4 – Les différents modes de tracé d'OpenGL

MCours.com