

Infographie - M1

Chapitre 2 - Tracés de segments et de cercles

V. Padovani

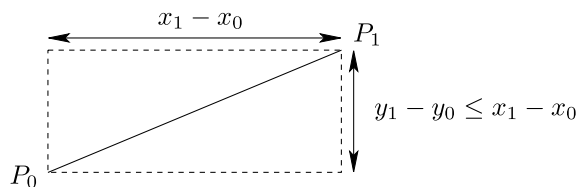
Equipe Preuves, Programmes et Systèmes, Université Paris 7

1 Tracés de segments

On considère le problème consistant à tracer à l'écran la représentation d'un segment allant d'un point $P_0 = (x_0, y_0)$ à un point $P_1 = (x_1, y_1)$, de manière à ce que les pixels allumés soient les plus proches possibles des points du segment "réel". On fera les hypothèses suivantes, auxquelles il est toujours possible de se ramener par échanges et/ou symétries :

- P_0 est à gauche de P_1 , i.e. $x_0 \leq x_1$,
- la *pente* du segment $\delta = (y_1 - y_0)/(x_1 - x_0)$ est comprise entre -1 et 1 .

On suppose donc que le rectangle contenant le segment $[P_0P_1]$ est plus large que haut. Lorsque $\delta \geq 0$, cette hypothèse correspond à la situation suivante :



Les segments de pixels représentant $[P_0P_1]$ seront donc tous horizontaux. On supposera de plus dans toute la suite que P_0 et P_1 sont à coordonnées *entières*. La pente de la droite est alors un *rationnel*, finiment représentable. Les algorithmes naïfs présentés ci-dessous n'exploitent pas cette hypothèse, et se contentent de calculer une approximation de cette pente à l'aide du type *double*. L'algorithme de Bresenham, lui, tire parti de cette hypothèse en n'utilisant que des variables entières : tous ses calculs sont donc exacts.

1.1 Algorithmes incrémental et multiplicatif

Voici d'abord deux méthodes naïves pour effectuer l'affichage de $[P_1P_2]$.

L'algorithme *incrémental* de tracé de segment peut être décrit par le pseudo-code suivant :

1. $\delta \leftarrow (y_1 - y_0)/(x_1 - x_0)$,
2. $x_c \leftarrow x_0$,
3. $y_c \leftarrow y_0$,
4. tant que $x_c \leq x_1$, faire :
 - (a) afficher le pixel le plus proche de (x_c, y_c) ,

$$(b) \ x_c \leftarrow x_c + 1,$$

$$(c) \ y_c \leftarrow y_c + \delta.$$

Le code ci-dessous utilise le type `double` pour représenter l'ordonnée du point courant.

```
// primitive d'affichage
void draw_pixel (int, int);

// arrondi d'un double a l'entier
// le plus proche
int round_int(double d) {
    return (int) (d + 0.5);
}

// algorithme incremental. suppose -1 <= d <= 1.
void line_add(int x0, int y0, int x1, int y1) {
    int xc;

    // y courant, initialise a y0
    double yc = y0;
    // calcul de la pente de la droite
    double d = ((double) (y1 - y0))/(x1 - x0);

    for (xc = x0; xc <= x1; xc++) {
        // affichage du pixel courant
        draw_pixel(xc, round_int(yc));
        // ajout de la pente au y courant
        yc = yc + d;
    }
}
```

Le problème posé par cette méthode est son manque de précision : en général, la valeur de d n'est qu'une approximation de la pente réelle du segment, et à chaque ajout de d à y_0 , la valeur de y_c s'écarte d'avantage de l'ordonnée du point d'abscisse x_c appartenant réellement au segment. On peut tenter de calculer une meilleure approximation des coordonnées de ce point en recalculant y_c à chaque itération, à partir de x , x_0 et de la pente de la droite :

```
// algorithme multiplicatif. suppose -1 <= d <= 1.
void line_mult(int x0, int y0, int x1, int y1) {
    int x, yc;
    // calcul de la pente de la droite
    double d = ((double) (y1 - y0))/(x1 - x0);

    for (x = x0; x <= x1; x++) {
        // calcul du y courant
```

```

        yc = round_int(y0 + (x - x0) * d);
        // affichage du pixel courant
        draw_pixel(x, yc);
    }
}

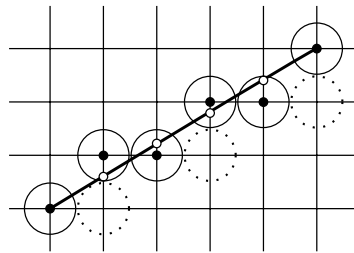
```

1.2 Algorithme de Bresenham

L'algorithme de Bresenham ou *algorithme de la ligne médiane*, élimine totalement le problème de l'imprécision des calculs, en ne se servant que de variables entières - cette possibilité existe en théorie, en remarquant que si les coordonnées des points sont entières, la pente de la droite est rationnelle, donc représentable de manière finitaire. On supposera cette pente de la droite est comprise entre 0 et 1.

Voici tout d'abord la version "réelle" de l'algorithme, inexacte, mais servant de point de départ à sa version optimisée. La boucle principale de cette fonction parcourt tous les x de x_0 à x_1 . On garde en mémoire dans une variable e la différence entre l'ordonnée courante (yc) et l'ordonnée du point du segment d'abscisse x .

La variable e est aussi appelée *variable de décision* : c'est à partir de sa valeur que l'on décide ou non d'incrémenter l'ordonnée courante. Dans la version ci-dessous, on incrémente yc dès que e mise à jour dépasse $1/2$.



```

// version réelle de l'algorithme de Bresenham.
// suppose x0 <= x1 et 0 <= dy/dx <= 1.

void mid_double (int x0, int y0, int x1, int y1) {
    int x, yc = y0, dx = x1 - x0, dy = y1 - y0;

    // apres chaque iteration, e vaut l'erreur entre yc et
    // l'ordonnee du point du segment d'abscisse x
    double e = 0;
    for (x = x0; x <= x1; x++) {
        draw_pixel(x, yc);
        // si l'erreur mise a jour depasse 1/2, on incremente yc
        // et on retranche 1 a l'erreur mise a jour
        if (e + ((double) dy / dx) > 1/2) {

```

```

        e = e + ((double) dy / dx) - 1;
        y++;
    }
    else e = e + ((double) dy / dx);
}
}

```

Voici à présent comment on obtient la version *discrète* de l'algorithme de la ligne médiane. Dans le programme initial, on a :

```

int dx = x1 - x0;
int dy = y1 - y0;
int e = 0;
...
if (e + ((double) dy / dx) > 1/2) {
    e = e + ((double) dy / dx) - 1;
    y++;
}
else e = e + ((double) dy / dx);
...

```

On effectue le changement de variable suivant (ed pour “erreur discrète”) :

- $ed \leftarrow (2 * dx * e) + (2 * dy) - dx$

On a alors :

- L'initialisation $e = 0$ devient $ed = (2 * dy) - dx$.
- Le test $(e + ((double) dy / dx) > 1/2)$ devient :
 - $2 * e + 2 * ((double) dy / dx) > 1$
 - $(2 * dx * e) + (2 * dy) > dx$
 - $(2 * dx * e) + (2 * dy) - dx > 0$
 - $ed > 0$
- Puisque ed vaut $2 * dx * e + 2 * dy - dx$, ajouter dy / dx à e revient à ajouter $(2 * dx) * (dy / dx)$ à ed, soit encore à lui ajouter $2 * dy$. L'affectation $e = e + ((double) dy / dx)$ devient donc :
 - $ed = ed + (2 * dx) * ((double) dy / dx)$
 - $ed = ed + (2 * dy)$
- Pour les mêmes raisons, $e = e + ((double) dy / dx) - 1$ devient :
 - $ed = ed + (2 * dx) * (((double) dy / dx) - 1)$
 - $ed = ed + (2 * dy) - (2 * dx)$

Voici à présent, avec ce changement de variable, la version discrète de l'algorithme de la ligne médiane, c'est-à-dire l'algorithme de Bresenham. Les deux incréments possibles pour `ed` sont pré-calculés, afin de minimiser encore le nombre d'opérations :

```
// Algorithme de Bresenham.
// suppose x0 <= x1 et 0 <= m <= 1.

void bresenham(int x0, int y0, int x1, int y1) {
    int x, y = y0;

    int dx = x1 - x0;
    int dy = y1 - y0;

    int ed = 2*dy - dx;
    int incr1 = 2*dy - 2*dx;
    int incr2 = 2*dy;

    for (x = x0; x <= x1; x++) {
        draw_pixel(x, y);
        if (ed > 0) {
            ed = ed + incr1;
            y++;
        }
        else ed = ed + incr2;
    }
}
```

2 Tracés de cercles

Le problème du tracé de cercles admet plusieurs solutions naïves, récursives ou non, basées sur le calcul de sinus et de cosinus. Ces solutions sont à la fois imprécises et gourmandes en temps de calcul. L'algorithme ci-dessous, dû à Bresenham, résout ces deux problèmes en ne considérant que des variables entières et des opérations élémentaires - il faut bien sûr supposer que les coordonnées du centre du cercle et son rayon sont entiers.

2.1 Algorithme de Bresenham pour les cercles

La méthode suivante permet de dessiner le second octant (entre les positions "12h" et "13h30") du cercle de rayon entier R centré en $(0,0)$. Le traitement général s'obtient par symétries et translations. Noter que cette version de l'algorithme de Bresenham est encore optimisable, mais ses versions optimales sont plus difficilement lisibles.

On part du pixel $(x, y) = (0, R)$. A chaque étape, on allume le pixel (x, y) . Le couple (x, y) est ensuite transformé en $(x + 1, y)$ avec $y = y$ (point "haut")

ou $y = y - 1$ (point “bas”). Le traitement est répété tant qu’on a $y > x$. Pour déterminer laquelle des deux ordonnées choisir, on garde en mémoire les deux quantités suivantes (h pour “haut”, b pour “bas”) :

- $\Delta_h(x, y) = (x + 1)^2 + y^2 - R^2$,
- $\Delta_b(x, y) = (x + 1)^2 + (y - 1)^2 - R^2$.

Voici un peu plus précisément à quoi correspondent ces deux quantités. Le point *réel* du cercle d’abscisse $x + 1$ est de coordonnées $(x + 1, y_r)$ avec

$$(x + 1)^2 + y_r^2 - R^2 = 0$$

On donc :

$$\Delta_h(x, y) - y^2 + y_r^2 = 0$$

$$\Delta_h(x, y) = y^2 - y_r^2$$

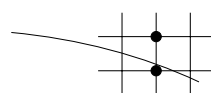
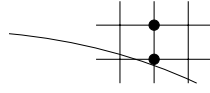
$$\Delta_b(x, y) - (y - 1)^2 + y_r^2 = 0$$

$$\Delta_b(x, y) = (y - 1)^2 - y_r^2$$

Autrement dit, $\Delta_h(x, y)$ et $\Delta_b(x, y)$ valent les erreurs signées entre les *carrés des ordonnées* des deux points candidats et celle du point appartenant réellement au cercle. Les propriétés suivantes, correspondant aux quatre situations représentées ci-dessous, sont facilement vérifiables :

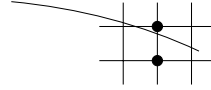
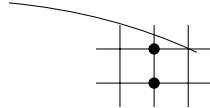
$$0 \leq \Delta_b(x, y)$$

$$0 \leq -\Delta_b(x, y) \leq \Delta_h(x, y)$$



$$\Delta_h(x, y) \leq 0$$

$$0 \leq \Delta_h(x, y) \leq -\Delta_b(x, y)$$



- $\Delta_b(x, y) \leq \Delta_h(x, y)$
- Si $0 \leq \Delta_b(x, y)$ alors :
 - $(x + 1, y - 1)$ est au dessus du point réel.
 - $(x + 1, y)$ est plus au dessus encore.
 - donc $(x + 1, y - 1)$ est le plus proche du point réel.
- Si $\Delta_h(x, y) \leq 0$ alors :
 - $(x + 1, y)$ est en dessous du point réel.

- $(x+1, y-1)$ est plus en dessous encore.
- donc $(x+1, y)$ est le point le plus proche du point réel.
- Si $\Delta_b(x, y) < 0 < \Delta_h(x, y)$, il y a trois possibilités :
 - $0 < -\Delta_b(x, y) < \Delta_h(x, y)$.

Dans ce cas, l'erreur entre le point $(x+1, y)$ et le point réel dépasse, en valeur absolue, celle entre le point $(x+1, y-1)$, et $(x+1, y-1)$ est le point le plus proche.

- $0 < \Delta_h(x, y) < -\Delta_b(x, y)$, et $(x+1, y)$ est le point le plus proche.
- $\Delta_h(x, y) = -\Delta_b(x, y)$. Par convention, on peut choisir $(x+1, y)$.

Autrement dit l'ordonnée du point le plus proche est y si $|\Delta_h(x, y)| < |\Delta_b(x, y)|$, et $y+1$ si $|\Delta_h(x, y)| > |\Delta_b(x, y)|$ - en cas d'égalité, les deux points candidats sont à égale distance du point réel, et l'on peut choisir l'un ou l'autre.

On remplace (x, y) par (x, y) . Les nouvelles valeurs de Δ_h et Δ_b peuvent être recalculées comme suit :

- Si $(x, y) = (x+1, y)$ on a :

$$\Delta_h(x+1, y) - \Delta_h(x, y) = (x+2)^2 - (x+1)^2 = 2x+3$$

$$\Delta_b(x+1, y) - \Delta_b(x, y) = (x+2)^2 - (x+1)^2 = 2x+3$$

donc

$$\Delta_h(x, y) = \Delta_h(x, y) + 2x+3$$

$$\Delta_b(x, y) = \Delta_b(x, y) + 2x+3.$$

- Sinon $(x, y) = (x+1, y-1)$ et :

$$\begin{aligned} \Delta_h(x+1, y-1) - \Delta_h(x, y) &= (x+2)^2 - (x+1)^2 + (y-1)^2 - y^2 \\ &= (2x+3) - 2y+1 \\ &= 2x-2y+4 \end{aligned}$$

$$\begin{aligned} \Delta_b(x+1, y-1) - \Delta_b(x, y) &= (x+2)^2 - (x+1)^2 + (y-2)^2 - (y-1)^2 \\ &= (2x+3) - 2y+3 \\ &= 2x-2y+6 \end{aligned}$$

donc

$$\Delta_h(x, y) = \Delta_h(x, y) + 2(x-y) + 4.$$

$$\Delta_b(x, y) = \Delta_b(x, y) + 2(x-y) + 6.$$