Bresenham

David Odin

Forma3Dev pour CPE–Lyon

2011

Bresenham – Plan

Qu'est-ce qu'une image ?

D'après WikiPedia : "*Une image (2D) est une fonction de RxR dans R où le couplet d'entrée est considéré comme une position spatiale, le singleton de sortie comme un codage.*"

Qu'est-ce qu'une image ?

D'après WikiPedia : "*Une image (2D) est une fonction de RxR dans R où le couplet d'entrée est considéré comme une position spatiale, le singleton de sortie comme un codage.*"

$$couleur = f(x, y)$$

$$f(x, y) = couleur$$

Oui, mais en Vrai ?

Pour nos besoins (et dans la plupart des cas), on utilisera :

- Un tableau 2D (`int image[hauteur][largeur];`)

Oui, mais en Vrai ?

Pour nos besoins (et dans la plupart des cas), on utilisera :

- Un tableau 2D (`int image[hauteur][largeur];`)
- Un tableau 1D (`int image[hauteur * largeur];`)

Oui, mais en Vrai ?

Pour nos besoins (et dans la plupart des cas), on utilisera :

- Un tableau 2D (`int image[hauteur][largeur];`)
- Un tableau 1D (`int image[hauteur * largeur];`)

La valeur stockée en chaque position peut être de différente nature :

- un booléen

Oui, mais en Vrai ?

Pour nos besoins (et dans la plupart des cas), on utilisera :

- Un tableau 2D (`int image[hauteur][largeur];`)
- Un tableau 1D (`int image[hauteur * largeur];`)

La valeur stockée en chaque position peut être de différente nature :

- un booléen
- un entier (niveau de gris)

Oui, mais en Vrai ?

Pour nos besoins (et dans la plupart des cas), on utilisera :

- Un tableau 2D (`int image[hauteur][largeur];`)
- Un tableau 1D (`int image[hauteur * largeur];`)

La valeur stockée en chaque position peut être de différente nature :

- un booléen
- un entier (niveau de gris)
- une couleur (une structure)

Oui, mais en Vrai ?

Pour nos besoins (et dans la plupart des cas), on utilisera :

- Un tableau 2D (`int image[hauteur][largeur];`)
- Un tableau 1D (`int image[hauteur * largeur];`)

La valeur stockée en chaque position peut être de différente nature :

- un booléen
- un entier (niveau de gris)
- une couleur (une structure)
- etc.

Bresenham – Plan

Effacer l'image

Effacer l'image, c'est simplement mettre la même valeur dans tout le tableau image.

```
int i, j;

for (i = 0; i < hauteur; i++)
  for (j = 0; j < largeur; j++)
    image[i][j] = 0;
```

Effacer l'image

Effacer l'image, c'est simplement mettre la même valeur dans tout le tableau image.

```
int i, j;

for (i = 0; i < hauteur; i++)
  for (j = 0; j < largeur; j++)
    image[i][j] = 0;
```

ou

```
int i;

for (i = 0; i < largeur * hauteur; i++)
  image[i] = 0;
```

Effacer l'image

Effacer l'image, c'est simplement mettre la même valeur dans tout le tableau image.

```
int i, j;

for (i = 0; i < hauteur; i++)
  for (j = 0; j < largeur; j++)
    image[i][j] = 0;
```

ou

```
int i;

for (i = 0; i < largeur * hauteur; i++)
  image[i] = 0;
```

ou

```
memset (image, 0, sizeof image);
```

Afficher un point

C’est encore plus simple :

- image[y][x] = 1;

Afficher un point

C’est encore plus simple :

- image[y][x] = 1;
- image[y $*$ largeur + x] = 1

UNE DROITE HORIZONTALE

Une simple boucle suffit pour tracer une droite entre $(x1, y)$ et $(x2, y)$ (en supposant que $x1 < x2$) :

```
int x;

for (x = x1; x <= x2; x++)
  image [y * largeur + x] = 1;
```

UNE DROITE HORIZONTALE

Une simple boucle suffit pour tracer une droite entre $(x1, y)$ et $(x2, y)$ (en supposant que $x1 < x2$) :

```
int x;

for (x = x1; x <= x2; x++)
  image [y * largeur + x] = 1;
```

ou

```
memset (image + y * largeur + x1, 1, (x2 - x1 + 1) * sizeof (int));
```

UNE DROITE VERTICALE

C'est tout aussi facile (avec $y1 < y2$) :

```
int y;

for (y = y1; y <= y2; y++)
  image [y * largeur + x] = 1;
```

Une droite verticale

C'est tout aussi facile (avec $y1 < y2$) :

```
int y;

for (y = y1; y <= y2; y++)
  image [y * largeur + x] = 1;
```

Mais malheureusement impossible avec memset...

Un rectangle

Un rectangle est simplement une combinaison de droites horizontales et verticales.

Une droite diagonale

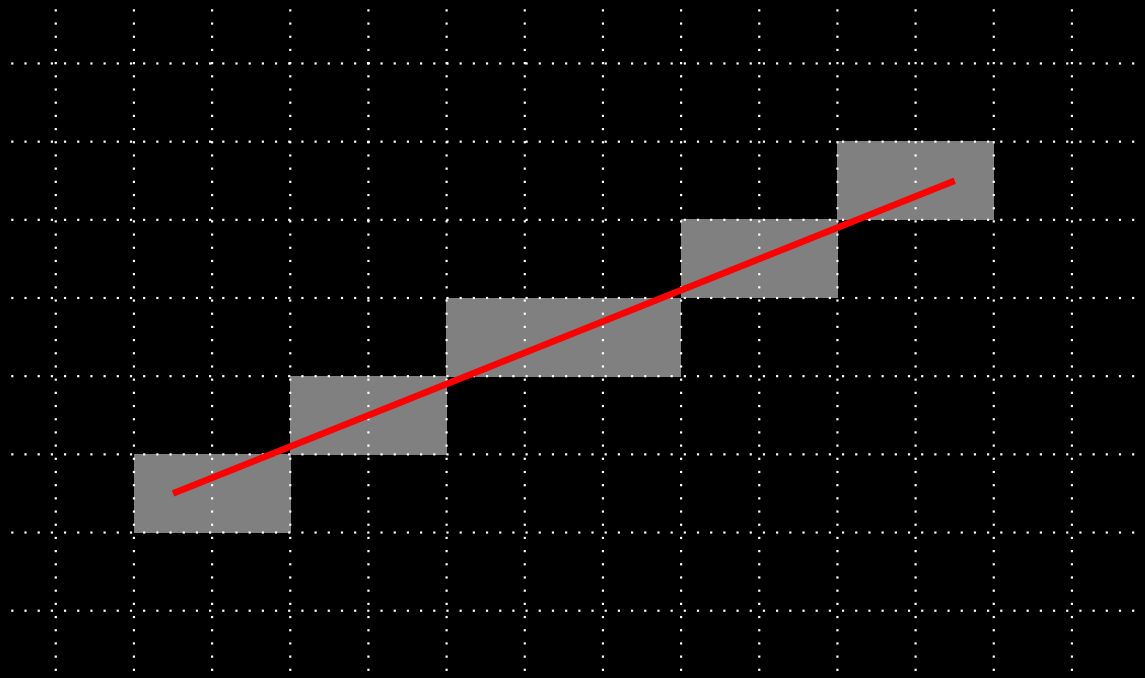
Une droite de pente 1 ou -1 peut facilement être obtenue de la même façon, avec une boucle simple. Mais l'intérêt est moins évident.

Bresenham – Plan

Bresenham – Plan

Pas si facile

Tracer une droite entre $(x1, y1)$ et $(x2, y2)$ quelconques est moins évident qu'il n'y parait.



Méthode naïve

Pour tracer une ligne plutôt horizontale ($|x2 - x1| >= |y2 - y1|$) :

```
int dx = x2 - x1;
int dy = y2 - y1;
float m = dy / dx; /* 0 < m <= 1 */
int x;

for (x = x1; x <= x2; x++)
  {
    int y = m * (x - x1) + y1 + 0.5;
    image [y * largeur + x] = 1;
  }
```

Améliorations

Le *y* calculé est toujours faux. Mais on peut savoir de combien il est faux, et réagir en fonction. Pour cela, on va stoquer dans une variable *erreur* l'erreur commise sur *y*. Et on s'arrangera pour qu'elle soit toujours inférieure à 1.

```
int dy = y2 - y1;
int dx = x2 - x1;
int y = y1;
float erreur = 0.0;
float m = dy / dx;
for (x = x1; x <= x2; x++)
  {
    image [y * largeur + x] = 1;
    erreur += m;
    if (erreur >= 0.5)
      {
        y++;
        erreur -= 1.0;
      }
  }
```

Encore mieux

Le problème est le calcul en flottant. Utilisons des entiers. Pour cela, trois astuces:

- Décaler l'erreur de 0.5 (on a alors une comparaison avec 0 au lieu de 0.5)

Encore mieux

Le problème est le calcul en flottant. Utilisons des entiers. Pour cela, trois astuces:

- Décaler l'erreur de 0.5 (on a alors une comparaison avec 0 au lieu de 0.5)
- Tout multiplier par dx (l'erreur peut alors être incrémentée par valeur entière)

Encore mieux

Le problème est le calcul en flottant. Utilisons des entiers. Pour cela, trois astuces:

- Décaler l'erreur de 0.5 (on a alors une comparaison avec 0 au lieu de 0.5)
- Tout multiplier par dx (l'erreur peut alors être incrémentée par valeur entière)
- Tout multiplier par 2 (pour eviter les 0.5 qui trainent)

Encore mieux

Le problème est le calcul en flottant. Utilisons des entiers. Pour cela, trois astuces:

- Décaler l'erreur de 0.5 (on a alors une comparaison avec 0 au lieu de 0.5)
- Tout multiplier par dx (l'erreur peut alors être incrémentée par valeur entière)
- Tout multiplier par 2 (pour eviter les 0.5 qui trainent)

Encore mieux

Le problème est le calcul en flottant. Utilisons des entiers. Pour cela, trois astuces:

- Décaler l'erreur de 0.5 (on a alors une comparaison avec 0 au lieu de 0.5)
- Tout multiplier par *dx* (l'erreur peut alors être incrémentée par valeur entière)
- Tout multiplier par 2 (pour eviter les 0.5 qui trainent)

Ce qui donne programme suivant :

```
int dy = y2 - y1;
int dx = x2 - x1;
int y = y1;
int erreur = -dx;
int m = 2 * dy;
for (x = x1; x <= x2; x++)
  {
    image [y * largeur + x] = 1;
    erreur += m;
    if (erreur >= 0)
      {
        y++;
        erreur -= 2 * dx;
      }
  }
```

Améliorations possibles

On peut toujours faire mieux :

- On peut sortir le 2 * dx de la boucle,

Améliorations possibles

On peut toujours faire mieux :

- On peut sortir le 2 * dx de la boucle,
- il vaut mieux traiter les horizontales, les verticales et les diagonales à part,

Améliorations possibles

On peut toujours faire mieux :

- On peut sortir le 2 * dx de la boucle,
- il vaut mieux traiter les horizontales, les verticales et les diagonales à part,
- Pour les pentes faibles, on peut tracer des morceaux d'horizontales (décalage du problème)

Bresenham – Plan

TRIGO ?

Pour tracer un cercle, ça se complique. La méthode naïve utilise des fonctions trigonométriques :

```
float t;
for (t = 0; t <= 2 * M_PI; t += 0.01)
  {
    int x = centre_x + rayon * cos (t);
    int y = centre_y + rayon * sin (t);
    image [y * largeur + x] = 1;
  }
```

Mais le résultat est catastrophique en terme de temps, et la valeur 0.01 est complétement aléatoire.

Équation cartésienne

Les racines carrées étant un peu moins gourmandes en temps de calcul que les fonctions trigonométriques, on peut être tenté d'utiliser une représentation cartésienne $y - y_c = \sqrt{R^2 - (x - x_c)^2}$:

```
int x;

for (x = -rayon; x <= rayon; x++)
  {
    int y = centre_y + sqrt (rayon * rayon - (x - center_x) * (x - center_x));
    image [y * largeur + x] = 1;
  }
```

Malheureusement, des points sont oubliés et le temps de calcul est loin d'être optimal.

Bresenham pour les cercles

Améliorations :

- Tracer 8 huitièmes de cercle.

Bresenham pour les cercles

Améliorations :

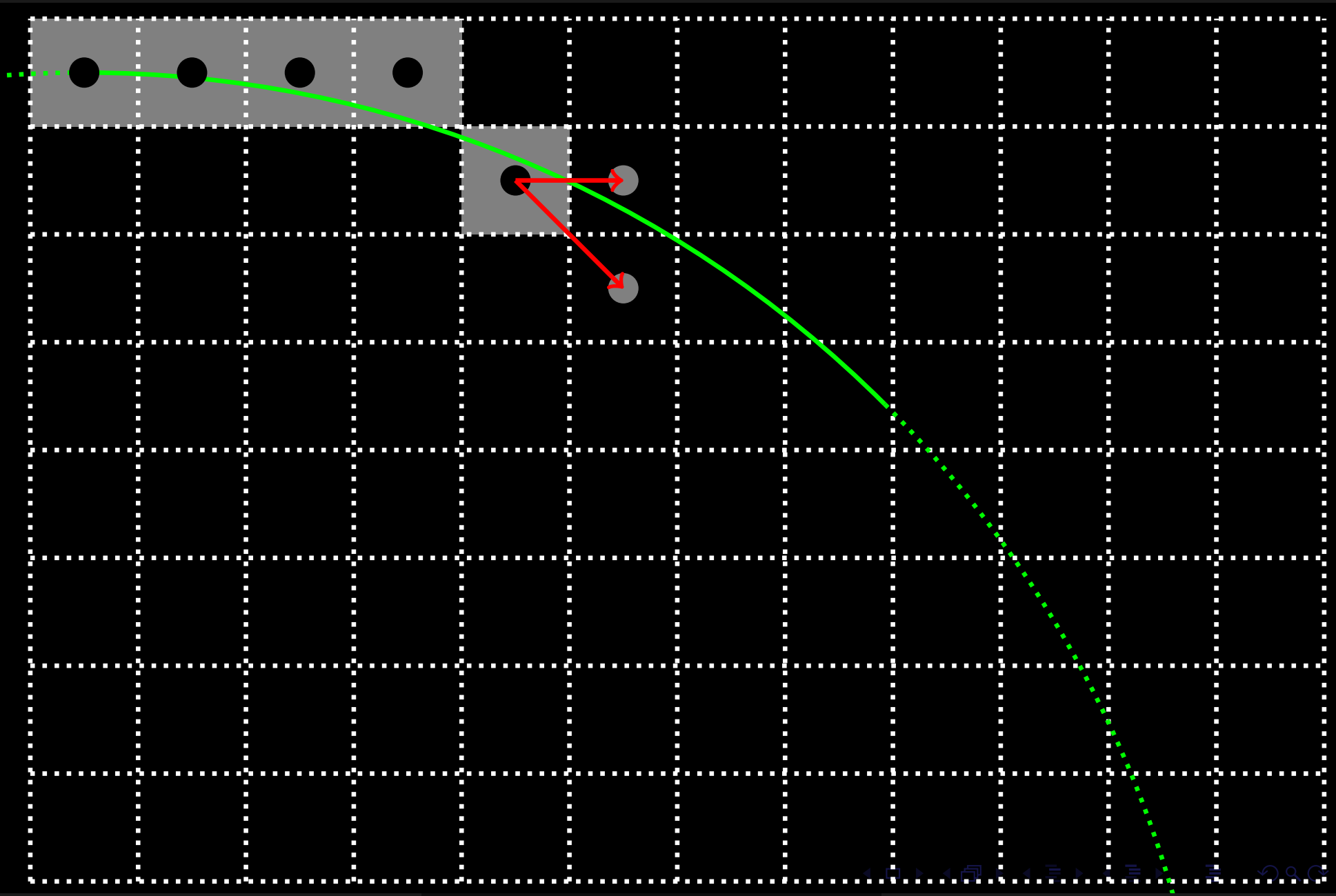
- Tracer 8 huitièmes de cercle.
- Utiliser l'idée de bresenham (signe de l'erreur commise).

Bresenham pour les cercles

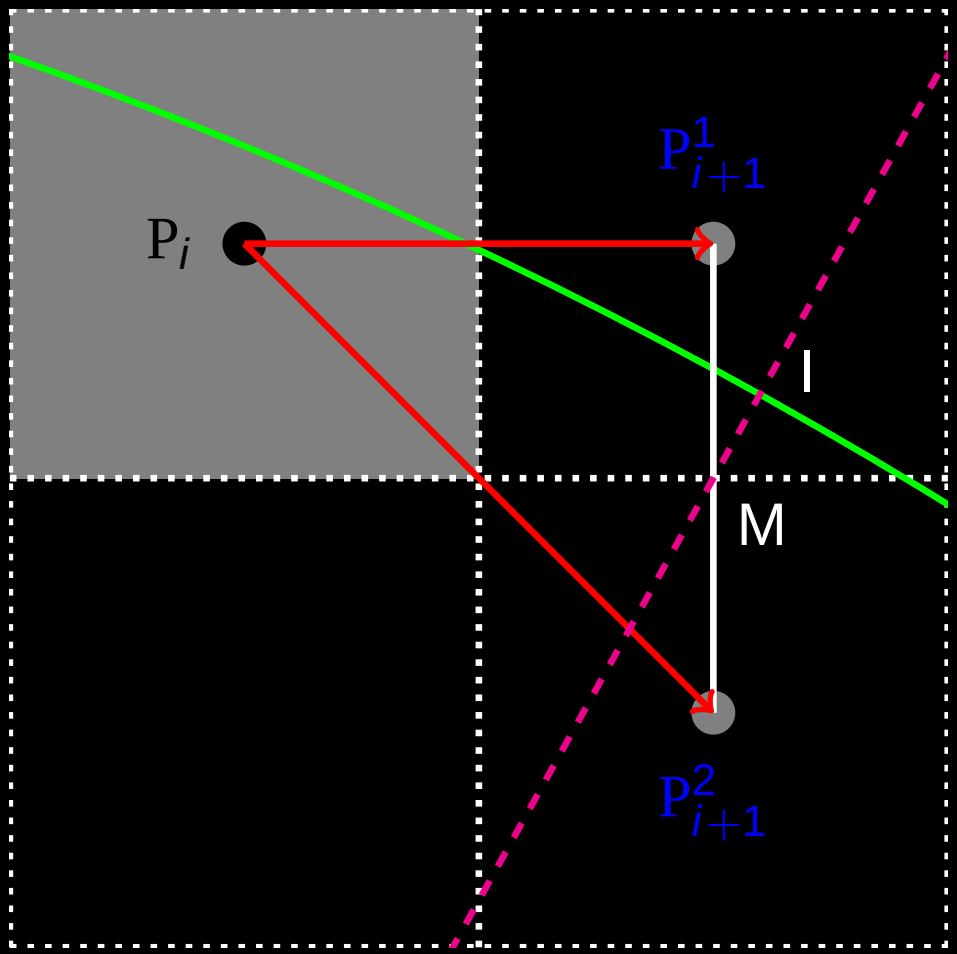
Améliorations :

- Tracer 8 huitièmes de cercle.
- Utiliser l'idée de bresenham (signe de l'erreur commise).
- N'utiliser que des entiers.

Erreur commise



Erreur commise



Calcul de l'erreur

$$m_i = x_M^2 + y_M^2 - R^2$$

Calcul de l'erreur

$$\begin{aligned} m_i &= x_M^2 + y_M^2 - R^2 \\ &= (x_i + 1)^2 + (y_i - 0,5)^2 - R^2 \end{aligned}$$

Calcul de l'erreur

$$\begin{aligned} m_i &= x_M^2 + y_M^2 - R^2 \\ &= (x_i + 1)^2 + (y_i - 0,5)^2 - R^2 \\ &= x_i^2 + y_i^2 + 2x_i - y_i + 1,25 - R^2 \end{aligned}$$

Calcul de l'erreur

$$\begin{aligned} m_i &= x_M^2 + y_M^2 - R^2 \\ &= (x_i + 1)^2 + (y_i - 0,5)^2 - R^2 \\ &= x_i^2 + y_i^2 + 2x_i - y_i + 1,25 - R^2 \end{aligned}$$

$$erreur_i = 4m_i = 4x_i^2 + 4y_i^2 + 8x_i - 4y_i + 5 - R^2$$

Progression de l'erreur

Cas 1

$x_{i+1} = x_i + 1, y_{i+1} = y_i$

$$erreur_{i+1} = 4m_{i+1}$$

Progression de l'erreur

Cas 1

$x_{i+1} = x_i + 1, y_{i+1} = y_i$

$$\begin{aligned} erreur_{i+1} &= 4m_{i+1} \\ &= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \end{aligned}$$

Progression de l'erreur

Cas 1

$x_{i+1} = x_i + 1, y_{i+1} = y_i$

$$\begin{aligned} erreur_{i+1} &= 4m_{i+1} \\ &= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\ &= 4(x_i + 1)^2 + 4y_i^2 + 8(x_i + 1) - 4y_i + 5 - R^2 \end{aligned}$$

Progression de l'erreur

Cas 1

$x_{i+1} = x_i + 1, y_{i+1} = y_i$

$$
\begin{aligned}
erreur_{i+1} &= 4m_{i+1} \\
&= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\
&= 4(x_i + 1)^2 + 4y_i^2 + 8(x_i + 1) - 4y_i + 5 - R^2 \\
&= 4x_i^2 + 8x_i + 4 + 4y_i^2 + 8x_i + 8 - 4y_i + 5 - R^2
\end{aligned}
$$

Progression de l'erreur

Cas 1

$x_{i+1} = x_i + 1, y_{i+1} = y_i$

$$
\begin{aligned}
erreur_{i+1} &= 4m_{i+1} \\
&= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\
&= 4(x_i + 1)^2 + 4y_i^2 + 8(x_i + 1) - 4y_i + 5 - R^2 \\
&= 4x_i^2 + 8x_i + 4 + 4y_i^2 + 8x_i + 8 - 4y_i + 5 - R^2 \\
&= \underbrace{4x_i^2 + 4y_i^2 + 8x_i - 4y_i + 5 - R^2}_{=4m_i} + 8x_i + 12
\end{aligned}
$$

Progression de l'erreur

Cas 1

$x_{i+1} = x_i + 1, y_{i+1} = y_i$

$$
\begin{aligned}
erreur_{i+1} &= 4m_{i+1} \\
&= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\
&= 4(x_i + 1)^2 + 4y_i^2 + 8(x_i + 1) - 4y_i + 5 - R^2 \\
&= 4x_i^2 + 8x_i + 4 + 4y_i^2 + 8x_i + 8 - 4y_i + 5 - R^2 \\
&= \underbrace{4x_i^2 + 4y_i^2 + 8x_i - 4y_i + 5 - R^2}_{=4m_i} + 8x_i + 12 \\
&= erreur_i + 8x_{i+1} + 4
\end{aligned}
$$

Progression de l'erreur

Cas 2

$x_{i+1} = x_i + 1, y_{i+1} = y_i - 1$

$erreur_{i+1} = 4m_{i+1}$

Progression de l'erreur

Cas 2

$x_{i+1} = x_i + 1, y_{i+1} = y_i - 1$

$$\begin{aligned} erreur_{i+1} &= 4m_{i+1} \\ &= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \end{aligned}$$

Progression de l'erreur

Cas 2

$x_{i+1} = x_i + 1, y_{i+1} = y_i - 1$

$$\begin{aligned} erreur_{i+1} &= 4m_{i+1} \\ &= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\ &= 4(x_i + 1)^2 + 4(y_i - 1)^2 + 8(x_i + 1) - 4(y_i - 1) + 5 - R^2 \end{aligned}$$

Progression de l'erreur

Cas 2

$x_{i+1} = x_i + 1, y_{i+1} = y_i - 1$

$$
\begin{aligned}
erreur_{i+1} &= 4m_{i+1} \\
&= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\
&= 4(x_i + 1)^2 + 4(y_i - 1)^2 + 8(x_i + 1) - 4(y_i - 1) + 5 - R^2 \\
&= 4x_i^2 + 8x_i + 4 + 4y_i^2 - 8y_i + 4 + 8x_i + 8 - 4y_i + 4 + 5 - R^2
\end{aligned}
$$

Progression de l'erreur

Cas 2

$x_{i+1} = x_i + 1, y_{i+1} = y_i - 1$

$$
\begin{aligned}
erreur_{i+1} &= 4m_{i+1} \\
&= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\
&= 4(x_i + 1)^2 + 4(y_i - 1)^2 + 8(x_i + 1) - 4(y_i - 1) + 5 - R^2 \\
&= 4x_i^2 + 8x_i + 4 + 4y_i^2 - 8y_i + 4 + 8x_i + 8 - 4y_i + 4 + 5 - R^2 \\
&= \underbrace{4x_i^2 + 4y_i^2 + 8x_i - 4y_i + 5 - R^2}_{=4m_i} + 8x_i - 8y_i + 20
\end{aligned}
$$

Progression de l'erreur

Cas 2

$x_{i+1} = x_i + 1, y_{i+1} = y_i - 1$

$$
\begin{aligned}
erreur_{i+1} &= 4m_{i+1} \\
&= 4x_{i+1}^2 + 4y_{i+1}^2 + 8x_{i+1} - 4y_{i+1} + 5 - R^2 \\
&= 4(x_i + 1)^2 + 4(y_i - 1)^2 + 8(x_i + 1) - 4(y_i - 1) + 5 - R^2 \\
&= 4x_i^2 + 8x_i + 4 + 4y_i^2 - 8y_i + 4 + 8x_i + 8 - 4y_i + 4 + 5 - R^2 \\
&= \underbrace{4x_i^2 + 4y_i^2 + 8x_i - 4y_i + 5 - R^2}_{=4m_i} + 8x_i - 8y_i + 20 \\
&= erreur_i + 8x_{i+1} - 8y_{i+1} + 4
\end{aligned}
$$

Cercles — implémentation.

```
int x = 0;
int y = rayon;
int erreur = 5 - 4 * rayon;

while (x <= y)
  {
    image [(y + centre_y) * largeur + x + centre_x] = 1; // +/- x et y et x<=>y
    x++;
    erreur += 8 * x + 4;
    if (erreur > 0)
      {
        y--;
        erreur -= 8 * y;
      }
  }
```

Cercle d'Andres

```
int x = 0;
int y = rayon;
int erreur = rayon - 1
while (x <= y)
  {
    image [(y + centre_y) * largeur + x + centre_x] = 1; // +/- x et y et x<=>y
    if (erreur >= 2 * x)
      {
        erreur = erreur - 2 * x - 1;
        x++;
      }
    else if (erreur <= 2 * (rayon - y))
      {
        erreur = erreur + 2 * y - 1;
        y--;
      }
    else
      {
        d = d + 2 * (y - x - 1);
        x++;
        y--;
      }
  }
```

Bresenham – Plan

Pourquoi ?

L'étape suivante est l'affichage d'un triangle. Plusieurs raisons à cela :

- Il s'agit d'une brique de base de la synthèse d'image,

Pourquoi ?

L'étape suivante est l'affichage d'un triangle. Plusieurs raisons à cela :

- Il s'agit d'une brique de base de la synthèse d'image,
- Beaucoup d'algorithmes produisent des triangles,

Pourquoi ?

L'étape suivante est l'affichage d'un triangle. Plusieurs raisons à cela :

- Il s'agit d'une brique de base de la synthèse d'image,
- Beaucoup d'algorithmes produisent des triangles,
- Si on sait afficher un triangle, on sait afficher beaucoup de choses.

Affichage d'un triangle rempli

L'algorithme d'affichage d'un triangle rempli est assez simple :

- Trouver le ymin et le ymax parmi les trois sommets.

Affichage d'un triangle rempli

L'algorithme d'affichage d'un triangle rempli est assez simple :

- Trouver le ymin et le ymax parmi les trois sommets.
- Initialiser deux tableaux : xmin et xmax, de taille (ymax - ymin + 1)

Affichage d'un triangle rempli

L'algorithme d'affichage d'un triangle rempli est assez simple :

- Trouver le ymin et le ymax parmi les trois sommets.
- Initialiser deux tableaux : xmin et xmax, de taille (ymax - ymin + 1)
- Utiliser l'algorithme de Bresenham pour les trois côtés, sans tracer les points, mais en mettant les tableau xmin et xmax à jour.

Affichage d'un triangle rempli

L'algorithme d'affichage d'un triangle rempli est assez simple :

- Trouver le ymin et le ymax parmi les trois sommets.
- Initialiser deux tableaux : xmin et xmax, de taille (ymax - ymin + 1)
- Utiliser l'algorithme de Bresenham pour les trois côtés, sans tracer les points, mais en mettant les tableau xmin et xmax à jour.
- Tracer chacune des lignes entre xmin et xmax.

Affichage d'un triangle rempli

L'algorithme d'affichage d'un triangle rempli est assez simple :

- Trouver le ymin et le ymax parmi les trois sommets.
- Initialiser deux tableaux : xmin et xmax, de taille (ymax - ymin + 1)
- Utiliser l'algorithme de Bresenham pour les trois côtés, sans tracer les points, mais en mettant les tableau xmin et xmax à jour.
- Tracer chacune des lignes entre xmin et xmax.

Affichage d'un triangle rempli

L'algorithme d'affichage d'un triangle rempli est assez simple :

- Trouver le ymin et le ymax parmi les trois sommets.
- Initialiser deux tableaux : xmin et xmax, de taille (ymax - ymin + 1)
- Utiliser l'algorithme de Bresenham pour les trois côtés, sans tracer les points, mais en mettant les tableau xmin et xmax à jour.
- Tracer chacune des lignes entre xmin et xmax.

Note: Cet algorithme permet aussi de tracer directement des polygônes convexes et même des polygônes horizontalement convexes.

Interpolation linéaire

On peut stocker plein de choses intéressantes dans les tableaux xmin et xmax, comme des valeurs de couleurs ou de normales interpolées entre les sommets du triangle. Ceci permet des effets comme l'ombrage de Gouraud ou celui de Phong.