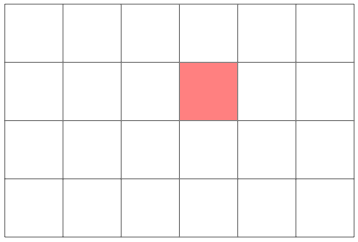
Pixels

Comment tracer, pixel par pixel, les figures géométriques de base ?

1. Pixels

1.1. L'unité d'affichage

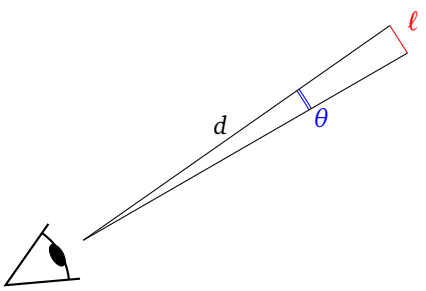
Le ***pixel*** est l'unité élémentaire d'affichage graphique numérique. Son abréviation est « px ». Le plus souvent le pixel est assimilé à un carré (ce sera le cas dans ce cours), mais peut aussi parfois être un rectangle.



Un pixel peut être noir ou blanc, mais aussi en couleur (le plus souvent obtenue par l'addition des couleurs de trois sous-pixels : un rouge, un vert, un bleu). La taille d'un pixel est généralement inférieure à 1 mm, elle dépend du type d'écran. Plus la taille d'un pixel est petite, plus la résolution est bonne. Une autre unité de mesure de la résolution est le ***nombre de pixels par pouce***, abrégé par « ppp » (ou *ppi* pour *pixels per inch*). Comme son nom l'indique, il s'agit de compter combien de pixels on peut juxtaposer sur une longueur d'un pouce (1 pouce = 25,4 mm). L'équivalent dans le monde de l'impression est le *dpi* (*dots per inch*). Voici quelques exemples approximatifs :

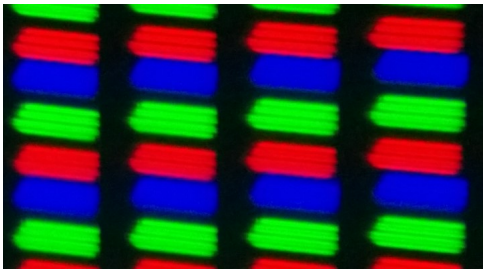
Type d'écran	Taille du pixel (mm)	Résolution (ppp)
Télévision	0.5	50
Ordinateur portable	0.17	150
Téléphone	0.05	500

L'œil humain a une acuité d'environ 1 minute d'arc (c'est-à-dire 1/60 de degré, soit $\theta = \frac{1}{60}\frac{2\pi}{360} \simeq 0.00029$ radian). Ainsi à une distance d, l'œil distingue des pixels d'une taille $\ell \simeq d\theta$ (avec ℓ et d en mètre et θ en radian).



Ainsi en regardant un écran à 20 cm de distance, augmenter la résolution au-delà de 400 ppp ne permet de ressentir une amélioration. Il faut alors mieux concentrer ses efforts sur la qualité de l'image (vitesse de rafraîchissement, couleurs, rayonnements…) et sur l'énergie (consommation, encre électronique…).

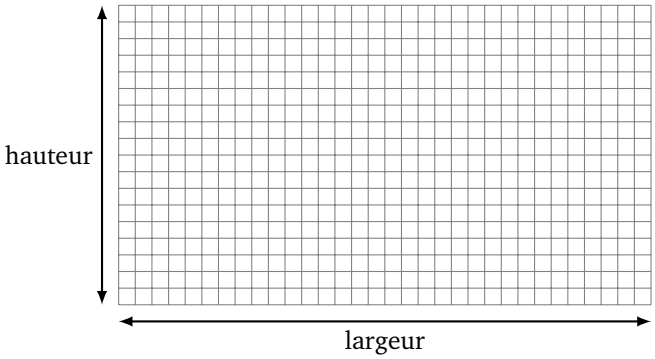
Un pixel n'est pas vraiment un petit carré ! Dans ce cours nous assimilerons un pixel à un (petit) carré. Cependant la réalité est plus compliquée. Un écran est composé d'une grille de sources lumineuses, appelées des *sous-pixels*. Chaque source illumine une petite zone qui n'est pas nécessairement un carré (cela peut être un disque ou un rectangle). Chaque source est souvent d'une seule couleur parmi rouge/vert/bleu, mais son intensité peut varier. Il n'y a donc physiquement aucun carré dans ce processus ! Sur la photo ci-dessous on pourrait regrouper trois sous-pixels superposés et appeler le petit carré correspondant un pixel, en lui attribuant la couleur résultant des trois sous-pixels. Il est plus juste de dire que les pixels sont un échantillonnage (relevé en des points d'une grille) du signal lumineux produit par l'écran.



Les sous-pixels d'un écran de téléphone portable (photo *Wikimedia*).

1.2. Écran

La ***taille*** d'un écran est le couple (largeur, hauteur) exprimé en pixels, habituellement notée sous la forme largeur × hauteur. Le résultat du produit correspond au nombre total de pixels de l'écran.



Le ***rapport d'image*** est le quotient de la largeur par la hauteur :

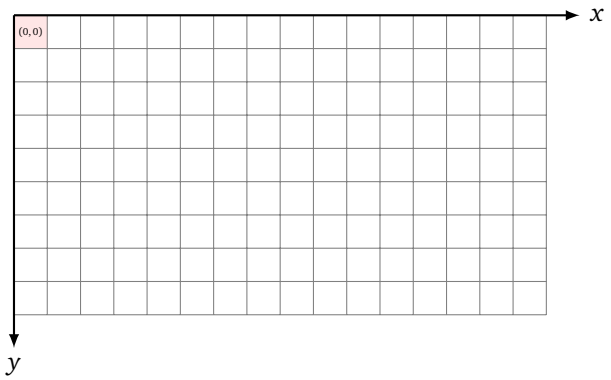
$$\text{rapport d'image} = \frac{\text{largeur}}{\text{hauteur}}.$$

Par exemple, pour un écran de taille 1024 × 768, cela signifie que chaque ligne contient 1 024 pixels et que chaque colonne contient 728 pixels. Le rapport d'image est

$$r = \frac{1024}{768} = \frac{4}{3} \simeq 1.33.$$

Nom du format	largeur	hauteur	rapport (fraction)	ratio (approché)
VGA	640	480	4/3	1.33
HD720	1280	720	16/9	1.77
HD1080	1920	1080	16/9	1.77
WUXGA	1920	1200	16/10	1.60
4K	3840	2160	16/9	1.77
8K	7680	4320	16/9	1.77
Image format Cinémascope	1024	430		2.38

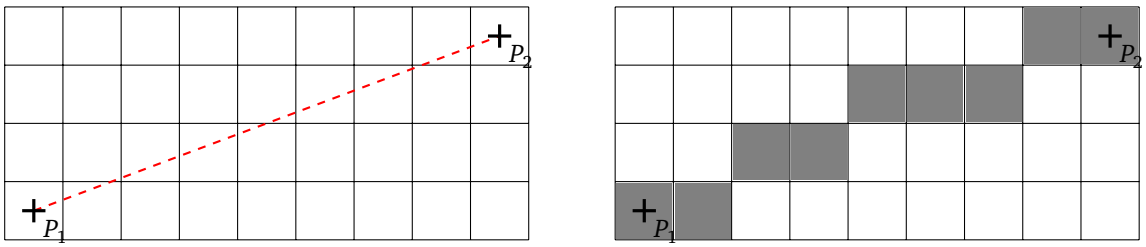
Le repère des pixels sur un écran a son axe des « y » dirigé vers le bas, contrairement à l'usage en mathématiques (que l'on suivra dans cet ouvrage). Ainsi le pixel en haut à gauche a pour coordonnées $(0,0)$.



2. Tracé de droites et de cercles

2.1. Tracé élémentaire d'un segment

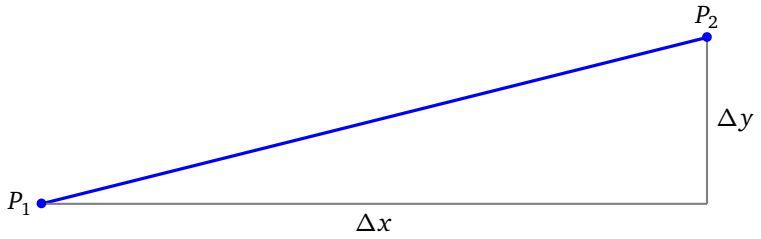
Le but est de tracer un segment composé de pixels entre deux points $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$. Nous supposons que ces points ont des coordonnées entières ($x_1, y_1, x_2, y_2 \in \mathbb{Z}$) et qu'un pixel est un carré de côté 1 centré en un couple d'entier (i, j).



L'idée naturelle est de calculer une équation de la droite (P_1P_2). Une équation réduite est $y = \alpha x + \beta$ où α est la pente et β l'ordonnée à l'origine. On va préférer écrire l'équation sous la forme :

$$y = \alpha(x - x_1) + y_1$$

où (x_1, y_1) est un point de la droite et la pente se calcule par $\alpha = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$.



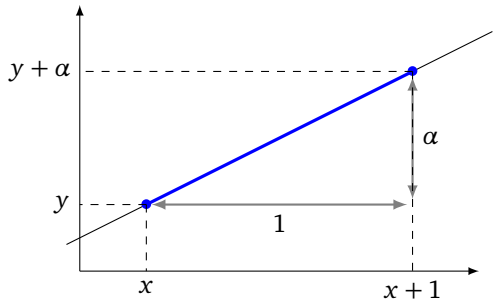
Hypothèse. On suppose $x_1 < x_2$, $y_1 < y_2$ et $\alpha = \frac{\Delta y}{\Delta x} \leqslant 1$.
Autrement dit, on trace un segment ayant une pente comprise entre 0 et 1, c'est-à-dire l'angle formé entre une horizontale et le segment est compris entre 0° et 45°. Les autres situations s'obtiennent par symétrie et ne seront pas détaillées ici.
Les pixels à colorier ont pour abscisses successives $i = x_1, x_1+1, \ldots, x_2$. L'ordonnée réelle correspondant à l'abscisse i est $y = \alpha(i - x_1) + y_1$ mais nous voulons une coordonnée entière qui s'obtient par un arrondi :

$$\boxed{j = \text{arrondi}(\alpha(i - x_1) + y_1)}$$

Remarque : si $\lfloor x \rfloor$ désigne la partie entière d'un réel x (obtenue par une commande `floor(x)` en *Python* par exemple) alors :

$$\text{arrondi}(x) = \lfloor x + \tfrac{1}{2} \rfloor$$

Pour minimiser les opérations on remarque que lorsque l'on passe de l'abscisse x à l'abscisse $x+1$ alors l'ordonnée passe de y à $y + \alpha$.



Algorithme (Algorithme du tracé élémentaire)**.**
Entrée : des entiers x_1, y_1, x_2, y_2 vérifiant l'hypothèse.
Sortie : le tracé des pixels reliant (x_1, y_1) à (x_2, y_2).

- Calculer $\alpha = \frac{y_2 - y_1}{x_2 - x_1}$.
- Poser $i = x_1$.
- Poser $y = y_1$.
- Tant que $i \leqslant x_2$:
  — $j = \text{arrondi}(y)$
  — afficher le pixel (i, j)
  — $i = i + 1$
  — $y = y + \alpha$

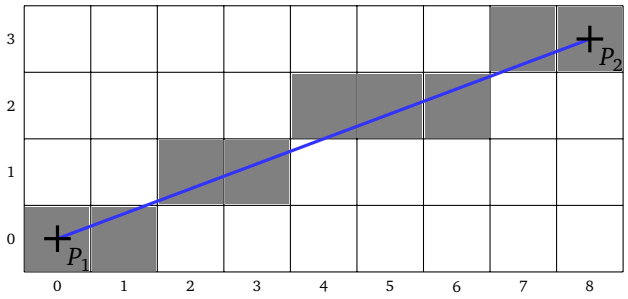
Exemple.
On souhaite tracer le segment reliant $P_1(0,0)$ et $P_2(8,3)$. On calcule :

$$\alpha = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{3}{8} = 0.375.$$

Voici les valeurs lorsqu'on applique l'algorithme :

i	y	j
0	0	0
1	0.375	0
2	0.75	1
3	1.125	1
4	1.5	2
5	1.875	2
6	2.25	2
7	2.625	3
8	3.0	3

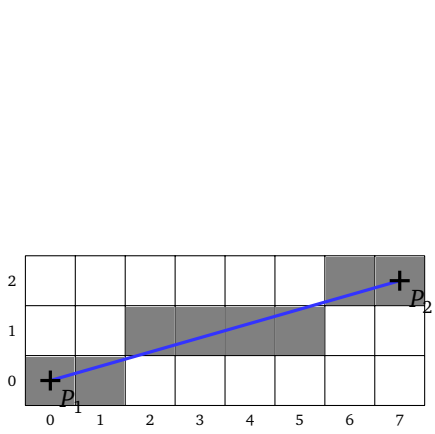
Ce qui donne :

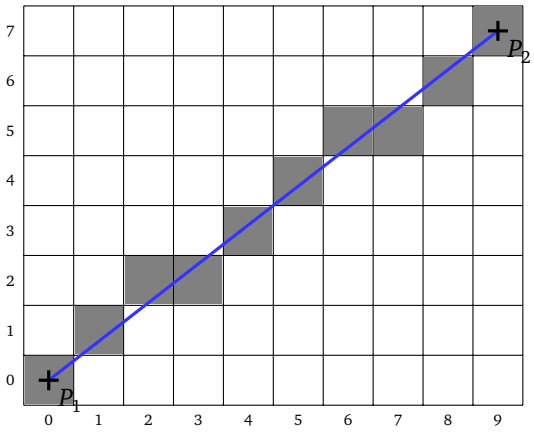


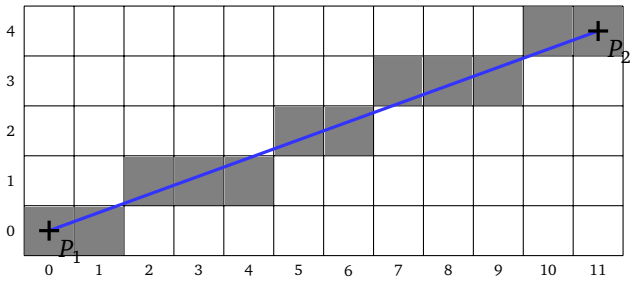
Noter que les calculs se font avec des nombres réels et sont donc assez lents.

2.2. Algorithme de Bresenham

Objectif : trouver un algorithme plus rapide que l'algorithme élémentaire en ne faisant que des calculs sur des entiers. Le résultat est équivalent à l'algorithme précédent mais beaucoup plus rapide.
Voici quelques tracés.



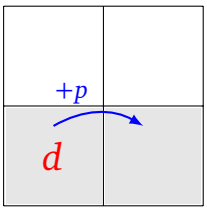




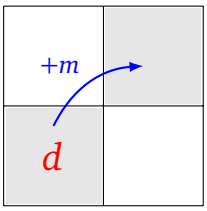
Algorithme (Algorithme de Bresenham)**.**
Entrée : des entiers x_1, y_1, x_2, y_2 vérifiant l'hypothèse.
Sortie : le tracé des pixels reliant (x_1, y_1) à (x_2, y_2).

- Calculer $p = 2\Delta y = 2(y_2 - y_1)$.
- Calculer $m = 2\Delta y - 2\Delta x = 2(y_2 - y_1) - 2(x_2 - x_1)$.
- Poser $d = 2\Delta y - \Delta x = 2(y_2 - y_1) - (x_2 - x_1)$.
- Poser $i = x_1$ et $j = y_1$.
- Tant que $i \leqslant x_2$:
  - afficher le pixel (i, j)
  - si $d < 0$:
    - $d = d + p$
  - sinon :
    - $j = j + 1$
    - $d = d + m$
  - $i = i + 1$

Noter que p, m sont des constantes entières avec $p > 0$ et $m < 0$; d est le « défaut », c'est une variable qui détermine s'il faut prendre le pixel juste à côté ou bien s'il faut monter d'un cran. Le défaut est alors mis à jour en lui ajoutant selon le cas une valeur fixe $p > 0$ ou $m < 0$ (si $d < 0$, le nouveau défaut $d + p$ a augmenté ; si $d \geqslant 0$, le nouveau défaut $d + m$ a diminué car $m < 0$).



Cas $d < 0$.



Cas $d \geqslant 0$.

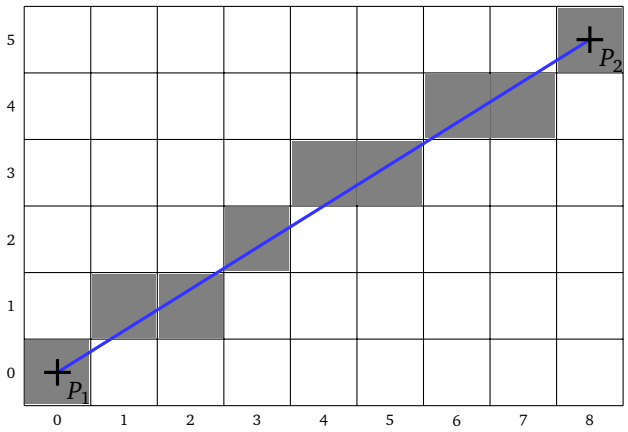
Noter qu'en plus les opérations sur les entiers sont élémentaires : ce sont des additions et un test de positivité.

Exemple.
On souhaite tracer le segment reliant $P_1(0,0)$ et $P_2(8,5)$. On calcule :

$$p = 10 \qquad m = -6 \qquad d_0 = 2$$

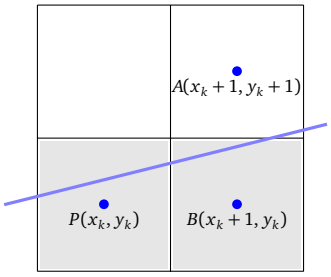
Voici les valeurs lorsqu'on applique l'algorithme :

i	j	d	monter ?	nouvelle valeur de d
0	0	2	oui	-4
1	1	-4	non	6
2	1	6	oui	0
3	2	0	oui	-6
4	3	-6	non	4
5	3	4	oui	-2
6	4	-2	non	8
7	4	8	oui	2
8	5			

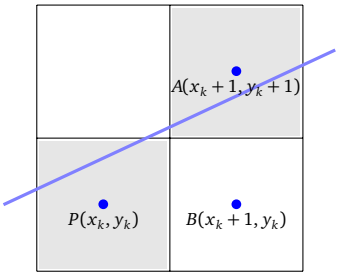


Preuve. Justifions que l'algorithme est correct.

- **Équation de la droite à coefficients entiers.** Nous avons vu qu'une équation de la droite (P_1P_2) est $y = \frac{\Delta y}{\Delta x}(x - x_1) + y_1$ avec $\Delta x = x_2 - x_1$ et $\Delta y = y_2 - y_1$. En multipliant cette équation par l'entier Δx on obtient l'équation :
$$(\Delta y)x - (\Delta x)y - (\Delta y)x_1 + (\Delta x)y_1 = 0$$
c'est-à-dire une équation $ax + by + c = 0$ avec a, b, c qui sont tous les trois des entiers.
Ainsi on définit la fonction :
$$E(x, y) = ax + by + c$$
où $a = \Delta y$, $b = -\Delta x$, $c = -(\Delta y)x_1 + (\Delta x)y_1$. Cette fonction détermine l'*écart* du point (x, y) par rapport à la droite (P_1P_2) :
— si $E(x, y) = 0$, le point (x, y) appartient à la droite,
— si $E(x, y) < 0$, le point (x, y) est situé au-dessus de la droite,
— si $E(x, y) > 0$, le point (x, y) est situé en-dessous de la droite.
- **Monter ou pas ?** Centrons en $P(x_k, y_k)$ un pixel déjà correctement colorié. Quels choix avons-nous pour colorier le pixel suivant ? Dans tous les cas le pixel suivant sera sur la colonne d'abscisse $x_k + 1$. Mais son ordonnée sera soit $y_k + 1$ (point A), soit y_k (point B).



Cas $D_k < 0$.



Cas $D_k \geqslant 0$.

Nous allons utiliser notre fonction d'écart E. Noter que l'on a toujours $E(A) \leqslant 0$ et $E(B) > 0$. On choisit entre A et B celui qui est le plus proche de la droite réelle, c'est-à-dire celui qui a le plus petit écart $|E(A)|$ ou $|E(B)|$ (en valeur absolue). Pour savoir lequel choisir on définit le *défaut* par
$$D(P) = E(A) + E(B).$$
On retient :
Si $D(P) \leqslant 0$ alors on choisit B, si $D(P) > 0$ alors on choisit A.
En effet, si $D(P) \leqslant 0$ alors $E(A) + E(B) \leqslant 0$ donc $E(B) \leqslant -E(A)$ mais comme $E(A) \leqslant 0$ et $E(B) > 0$ alors on a bien $|E(B)| \leqslant |E(A)|$.

- **Calcul du défaut.** Calculons D_k le défaut au point $P(x_k, y_k)$:

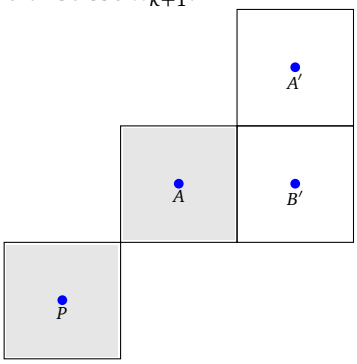
$$\begin{aligned} D_k = D(x_k, y_k) &= E(A) + E(B) = E(x_k + 1, y_k + 1) + E(x_k + 1, y_k) \\ &= a(x_k + 1) + b(y_k + 1) + c \quad + \quad a(x_k + 1) + by_k + c \\ &= 2ax_k + 2by_k + 2a + b + 2c \end{aligned}$$

  En particulier D_0 le défaut au point initial (x_1, y_1) est :

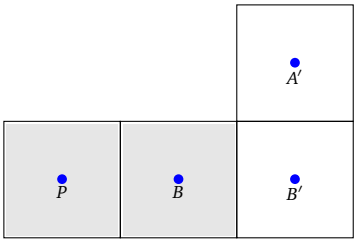
$$\begin{aligned} D_0 &= 2ax_1 + 2by_1 + 2a + b + 2c \\ &= 2(ax_1 + by_1 + c) + 2a + b \qquad (\star) \\ &= 2a + b \\ &= 2\Delta y - \Delta x \end{aligned}$$

  À la ligne $(\star)$ on a utilisé le fait que le point initial (x_1, y_1) est exactement sur la droite, donc $ax_1 + by_1 + c = 0$.

- **Formule de récurrence du défaut.**
  Nous allons maintenant trouver la relation qui lie le défaut d'un pixel D_k au défaut du pixel suivant D_{k+1}. On part toujours du point $P(x_k, y_k)$, on a déjà calculé D_k. Le calcul de D_{k+1} dépend du choix A ou B pour le pixel d'abscisse x_{k+1}.

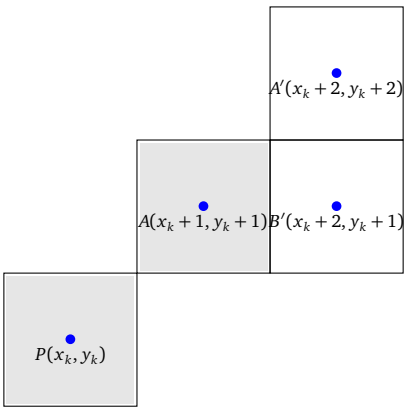


Cas A.



Cas B.

Cas du choix A (cas $D_k \geqslant 0$). Le pixel suivant a pour abscisse $x_k + 2$ et pour ordonnée soit $y_k + 2$ (A'), soit $y_k + 1$ (B').



Cas A.

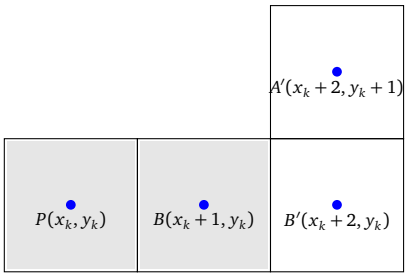
Calculons alors le défaut :

$$\begin{aligned} D_{k+1} = E(A') + E(B') &= E(x_k + 2, y_k + 2) + E(x_k + 2, y_k + 1) \\ &= a(x_k + 2) + b(y_k + 2) + c \quad + \quad a(x_k + 2) + b(y_k + 1) + c \\ &= 2ax_k + 2by_k + 4a + 3b + 2c \\ &= D_k + 2a + 2b \end{aligned}$$

Ainsi en posant $m = 2a + 2b = 2\Delta y - 2\Delta x$, on obtient la formule de récurrence :

$$D_{k+1} = D_k + m.$$

Cas du choix B (cas $D_k < 0$). Le pixel suivant a pour abscisse $x_k + 2$ et pour ordonnée soit $y_k + 1$ (A'), soit y_k (B').



Cas B.

Calculons alors le défaut :

$$\begin{aligned} D_{k+1} &= E(A') + E(B') = E(x_k+2, y_k+1) + E(x_k+2, y_k) \\ &= a(x_k+2) + b(y_k+1) + c \quad + \quad a(x_k+2) + by_k + c \\ &= 2ax_k + 2by_k + 4a + b + 2c \\ &= D_k + 2a \end{aligned}$$

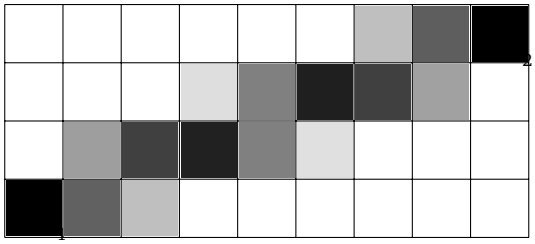
Ainsi en posant $p = 2a = 2\Delta y$, on obtient la formule de récurrence :

$$D_{k+1} = D_k + p.$$

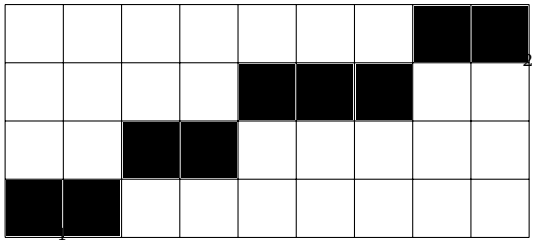
Conclusion : on sait calculer D_{k+1} qui permet de choisir entre A' et B'. Ainsi à partir de D_0 et par les formules de récurrence ci-dessus on calcule la suite des défauts D_k et on sait donc s'il faut monter le pixel d'un cran ou pas.

2.3. Anticrénelage (*antialiasing*)

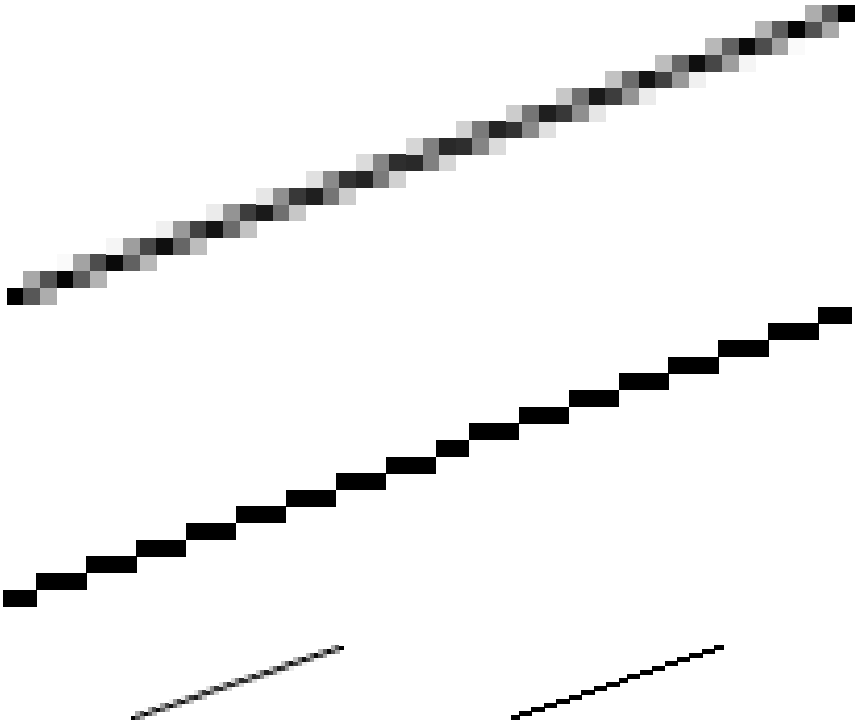
Le tracé d'un segment est nécessairement une approximation d'un tracé de droite, qui présente des défauts de visualisation appelée « crénelage ». On remédie à cette imperfection à l'aide d'un anticrénelage (*antialiasing*) qui colorie aussi le pixel laissé de côté (le pixel A ou B écarté dans les algorithmes précédents).



Avec anticrénelage



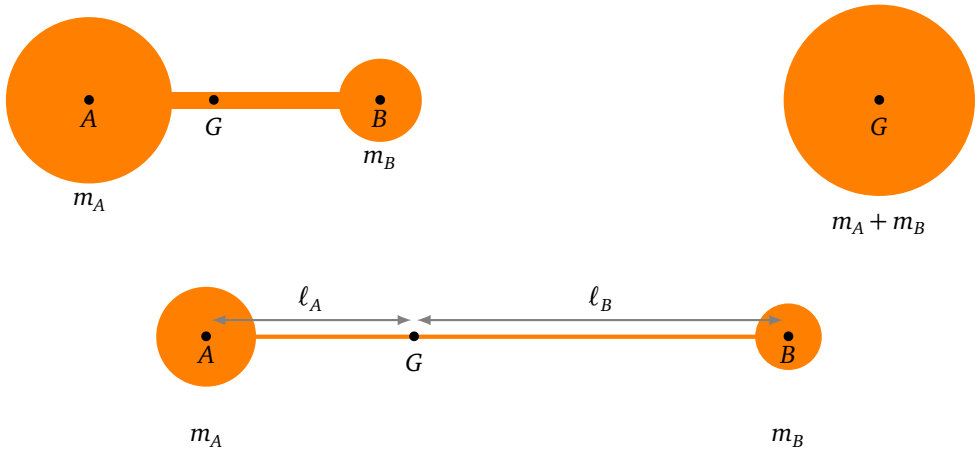
Sans anticrénelage



De près le rendu n'est pas impressionnant mais de loin l'amélioration est sensible.
Nous allons expliquer un algorithme simple pour dessiner un segment avec anticrénelage. Nous avons donc deux pixels l'un au-dessus de l'autre à colorier : le pixel A et le pixel B. Comment répartir 100% de couleur entre ces deux pixels ?
Commençons par expliquer une analogie issue de la physique. Considérons deux masses ponctuelles liées entre elles : m_A centrée en A et m_B centrée en B. Lorsque l'on regarde ce système d'assez loin on peut le considérer comme un système formé d'une seule masse ponctuelle : la masse totale est $m = m_A + m_B$ et est centrée au centre de gravité G. Ce centre de gravité G est défini par la relation :

$$\ell_A m_A = \ell_B m_B$$

où $\ell_A = AG$ et $\ell_B = BG$.



Reprenons cette idée : nous répartissons une masse totale $m = 1$ (100% de la couleur) centrée exactement sur la droite réelle en G entre deux points A et B.
Notons (x, y) le point G de la droite réelle. Soit alors :

$$j = \lfloor y \rfloor \qquad \text{et} \qquad \ell = y - \lfloor y \rfloor$$

respectivement la partie entière et la partie fractionnaire de y. La distance BG est ℓ et la distance AG est $1-\ell$ (car $AB=1$). Ainsi pour que les couleurs satisfassent la formule du centre de gravité il faut choisir $m_A=\ell$ et $m_B=1-\ell$ comme intensité de couleur respectivement pour A et B car on a alors bien $(1-\ell)\ell=\ell(1-\ell)$. On adapte l'algorithme du tracé élémentaire pour y ajouter l'anticrénelage.

Algorithme (Algorithme d'anticrénelage)**.**
Entrée : des entiers x_1, y_1, x_2, y_2 vérifiant l'hypothèse.
Sortie : le tracé des pixels reliant (x_1, y_1) à (x_2, y_2).

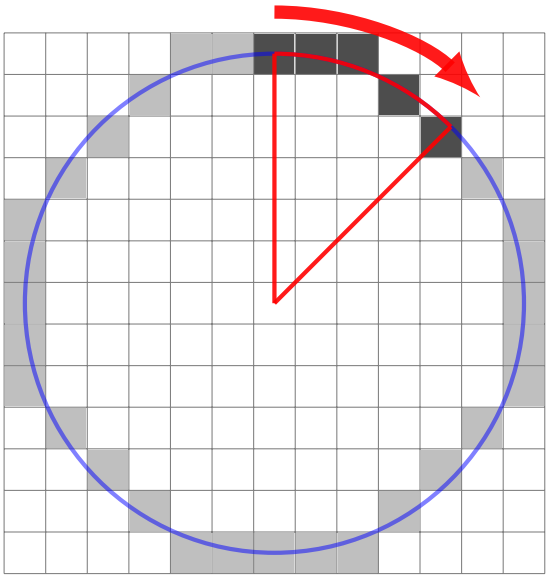
- Calculer $\alpha = \frac{y_2-y_1}{x_2-x_1}$.
- Poser $i = x_1$.
- Poser $y = y_1$.
- Tant que $i \leqslant x_2$:
  - $j = \lfloor y \rfloor$
  - $\ell = y - \lfloor y \rfloor$
  - afficher le pixel (i, j) avec l'intensité de couleur $1-\ell$
  - afficher le pixel $(i, j+1)$ avec l'intensité de couleur ℓ
  - $i = i+1$
  - $y = y+\alpha$

2.4. Tracé d'un cercle

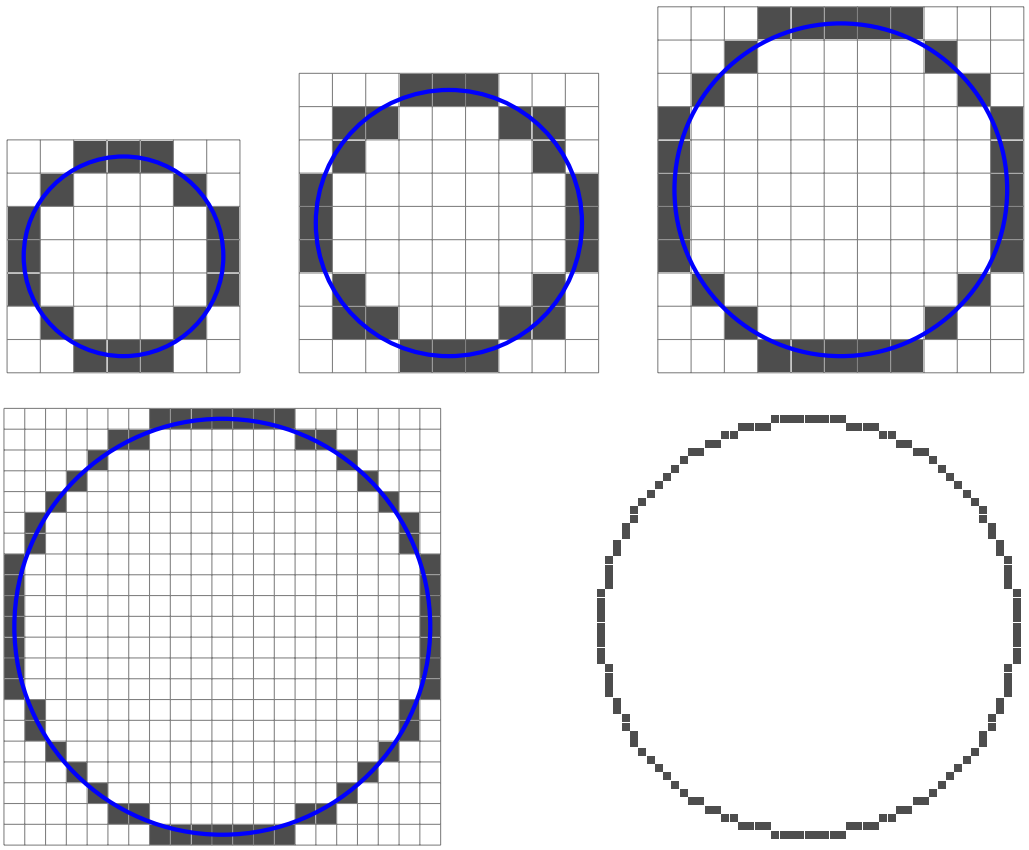
On souhaite tracer un cercle de rayon r centré en (x_0, y_0). Ce cercle a pour équation :

$$(x-x_0)^2+(y-y_0)^2=r^2.$$

Pour cela on va se contenter de tracer l'arc de cercle compris entre 90° et 45°. Le reste s'obtiendra par symétrie.



Voici quelques exemples.



Voici l'algorithme du tracé de l'arc de cercle de type Bresenham car il n'utilise que des entiers. Le point de départ est le point $(0, r)$ (point le plus haut du cercle).

Algorithme (Algorithme de tracé de cercle)**.**
Entrée : des entiers x_0, y_0 pour le centre et un entier r pour le rayon.
Sortie : le tracé d'un arc du cercle centré en (x_0, y_0) de rayon r.

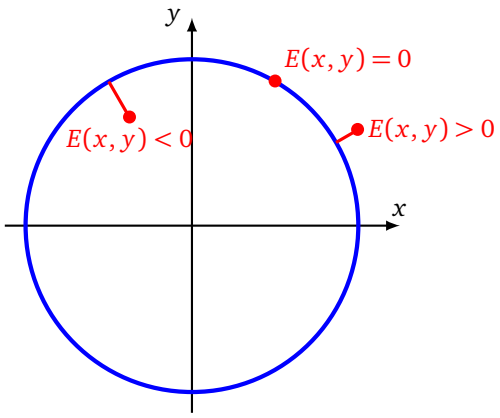
- Poser $i = 0$.
- Poser $j = r$.
- Poser $d = 3 - 2r$.
- Tant que $i \leqslant j$:
  - afficher le pixel $(x_0 + i, y_0 + j)$
  - si $d < 0$:
    - $d = d + 4i + 6$
  - sinon :
    - $d = d + 4i - 4j + 10$
    - $j = j - 1$
  - $i = i + 1$

Sur la portion tracée (de 90° à 45°) il y a exactement un pixel pour chaque i. Pour obtenir le cercle complet, en plus du pixel en $(x_0 + i, y_0 + j)$, il faut afficher par symétrie les pixels en $(x_0 + j, y_0 + i)$, $(x_0 - i, y_0 + j)$, $(x_0 - j, y_0 + i)$ $(x_0 + i, y_0 - j)$, $(x_0 + j, y_0 - i)$, $(x_0 - i, y_0 - j)$, $(x_0 - j, y_0 - i)$.
Preuve. La preuve est similaire à celle de l'algorithme du tracé d'un segment de Bresenham.

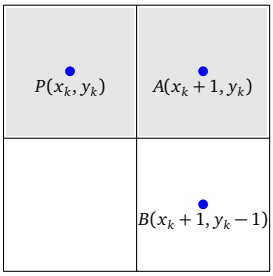
- **Équation du cercle.**
  On suppose que le cercle est centré à l'origine : $(x_0, y_0) = (0, 0)$. Un cercle quelconque s'obtiendrait par translation. Soit la fonction d'écart :

$$E(x, y) = x^2 + y^2 - r^2$$

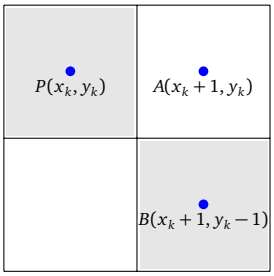
Si $E(x,y)=0$ le point (x,y) est sur le cercle, si $E(x,y)>0$ le point est à l'extérieur, si $E(x,y)<0$ le point est à l'intérieur.



- **Descendre ou pas ?**
  Pour un pixel centré en P déjà tracé, nous devons choisir pour le pixel suivant entre A et B. On choisit le point le plus proche du cercle, c'est-à-dire le minimum entre $|E(A)|$ ou $|E(B)|$. On définit le *défaut* par
  $$D(P)=E(A)+E(B)$$
  et on a :
  Si $D(P)<0$ alors on choisit A, si $D(P)\geqslant 0$ alors on choisit B.

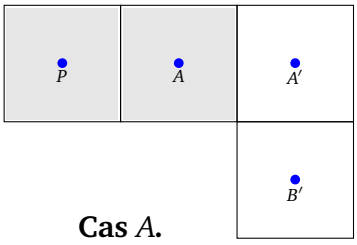


Cas $D_k<0$.

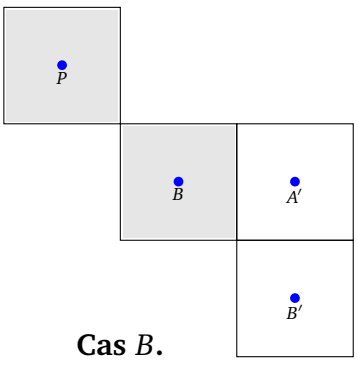


Cas $D_k\geqslant 0$.

- **Calcul du défaut.** Calculons D_k le défaut au point $P(x_k,y_k)$:
  $$\begin{aligned} D_k = D(x_k,y_k) &= E(A)+E(B) = E(x_k+1,y_k)+E(x_k+1,y_k-1)\\ &= (x_k+1)^2+y_k^2-r^2 \quad + \quad (x_k+1)^2+(y_k-1)^2-r^2\\ &= 2x_k^2+2y_k^2+4x_k-2y_k+3-2r^2 \end{aligned}$$
  En particulier D_0 le défaut au point initial $(0,r)$ vaut :
  $$D_0=3-2r.$$
- **Formule de récurrence du défaut.**
  On part du point $P(x_k,y_k)$, connaissant D_k on calcule D_{k+1} en fonction du choix A ou B pour le pixel d'abscisse x_{k+1}.

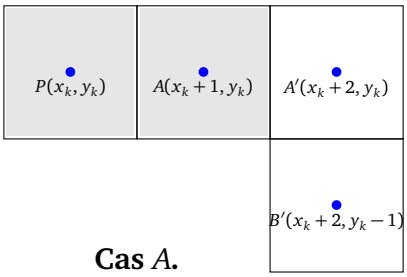


Cas A.



Cas B.

Cas du choix A (cas $D_k < 0$). Le pixel suivant a pour abscisse $x_k + 2$ et pour ordonnée soit y_k (A'), soit $y_k - 1$ (B').

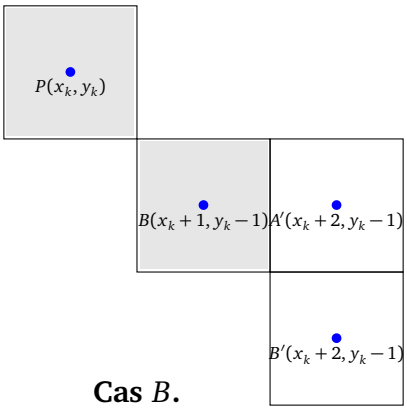


Cas A.

Le défaut se calcule ainsi :

$$\begin{aligned} D_{k+1} &= E(A') + E(B') = E(x_k+2, y_k) + E(x_k+2, y_k-1) \\ &= (x_k+2)^2 + y_k^2 - r^2 \quad + \quad (x_k+2)^2 + (y_k-1)^2 - r^2 \\ &= D_k + 4x_k + 6 \end{aligned}$$

Cas du choix B (cas $D_k \geqslant 0$). Le pixel suivant a pour abscisse $x_k + 2$ et pour ordonnée soit $y_k - 1$ (A'), soit $y_k - 2$ (B').



Cas B.

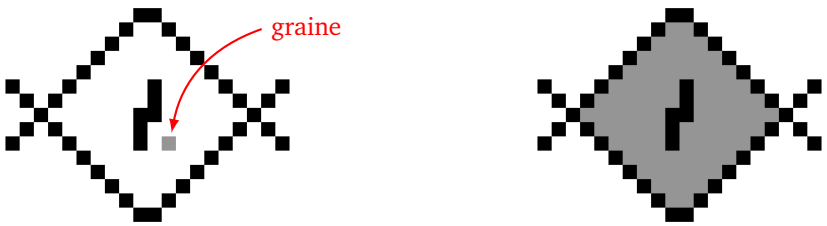
$$\begin{aligned} D_{k+1} &= E(A') + E(B') = E(x_k+2, y_k-1) + E(x_k+2, y_k-2) \\ &= (x_k+2)^2 + (y_k-1)^2 - r^2 \quad + \quad (x_k+2)^2 + (y_k-2)^2 - r^2 \\ &= D_k + 4x_k - 4y_k + 10 \end{aligned}$$

Conclusion : on sait calculer D_{k+1} qui permet de choisir entre A' et B'.

3. Colorier

3.1. Colorier par diffusion

Objectif. On considère une image formée par exemple de pixels noirs et blancs. On souhaite colorier en gris une zone de l'image. Pour cela, on part d'un pixel initial (la graine, *seed*) et il faut colorier en gris toute la zone de pixels blancs contenant la graine et bordée par des pixels noirs.

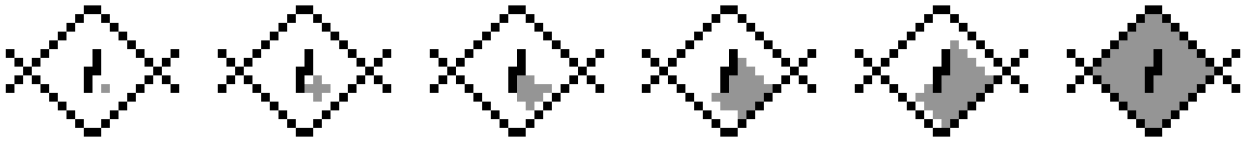


Principe. On part de la graine, qu'on colorie en gris. Pour chacun de ses quatre pixels voisins (gauche, bas, droite, haut), si c'est un pixel blanc, on applique le processus à partir de ce nouveau pixel.
Ci-dessous la graine et ses quatre voisins.

Mise en œuvre. Ce procédé peut être traduit en algorithmes par différentes méthodes : algorithme récursif, algorithme avec une pile ou algorithme avec une file. Dans tous les cas ce sont des algorithmes qui utilisent beaucoup de mémoire afin de stocker les pixels en attente d'être traités. On choisit un algorithme avec file (*queue* ou *fifo*, *first in, first out*) un peu moins gourmand en mémoire que les autres choix.

> **Algorithme** (Algorithme de remplissage par diffusion (*flood fill*))**.**
> **Entrée :** un tableau de pixels noirs ou blancs, un pixel initial (x_0, y_0) (la graine)
> **Sortie :** un tableau de pixels noirs, blancs ou gris
> - `file` $= [(x_0, y_0)]$
> - Tant que la `file` n'est pas vide :
> — prendre (x, y) un élément de `file` (et le retirer de la file)
> — si (x, y) est un pixel blanc :
> — le colorier en gris,
> — ajouter à la `file` les quatre voisins $(x-1, y)$, $(x, y+1)$, $(x+1, y)$, $(x, y-1)$.

Quelques étapes du processus :



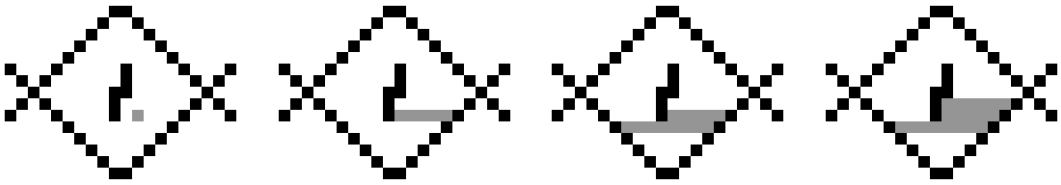
3.2. Colorier par balayage

Le coloriage par balayage en un peu plus performant au niveau de la mémoire car il colorie une portion de ligne avant d'ajouter les pixels au-dessus ou en-dessous dans la file d'attente.

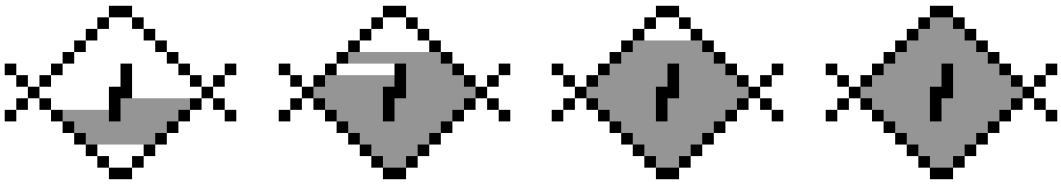
> **Algorithme** (Algorithme de balayage (*scan fill*))**.**
> **Entrée :** un tableau de pixels noirs ou blancs, un pixel initial (x_0, y_0) (la graine)
> **Sortie :** un tableau de pixels noirs, blancs ou gris
> - `file` $= [(x_0, y_0)]$
> - Tant que la `file` n'est pas vide :
> — prendre (x, y) un élément de `file` (et le retirer de la file)
> — poser $x_{\text{min}} = x - 1$
> — tant que (x_{min}, y) est un pixel blanc :
> — le colorier en gris,
> — faire $x_{\text{min}} = x_{\text{min}} - 1$.
> — poser $x_{\text{max}} = x$
> — tant que (x_{max}, y) est un pixel blanc :
> — le colorier en gris,
> — faire $x_{\text{max}} = x_{\text{max}} + 1$.
> — Pour chaque x' entre $x_{\text{min}} + 1$ et $x_{\text{max}} - 1$:

> — si $(x', y+1)$ est un pixel blanc, l'ajouter à la file,
> — si $(x', y-1)$ est un pixel blanc, l'ajouter à la file.

Ci-dessous la graine et les premières lignes.



Quelques étapes plus avancées :

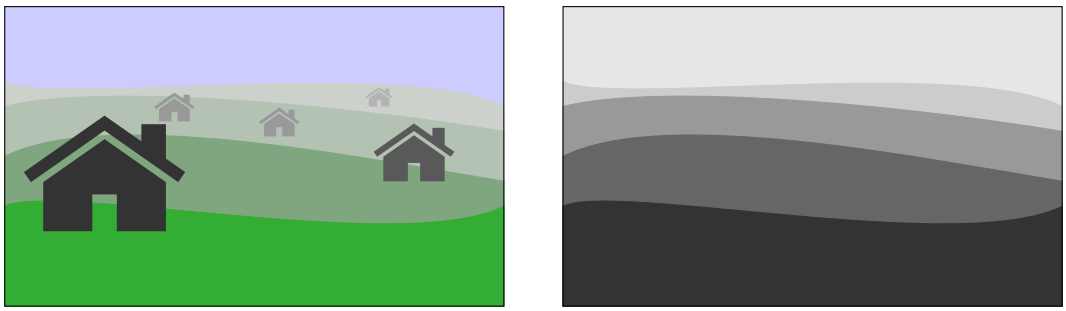


4. Animation

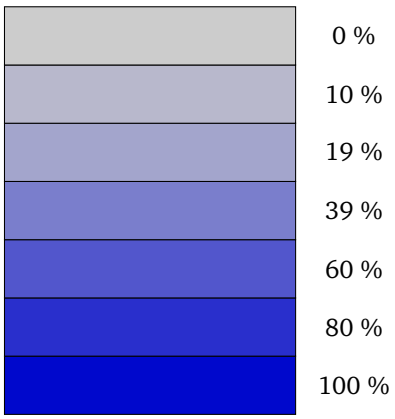
Nous expliquons quelques principes liés à l'animation des images. Les premiers dessins animés étaient basés sur des *calques* : une image fixe pour le calque de l'arrière-plan, un calque pour le premier plan avec par exemple un personnage à redessiner de nombreuses fois afin de simuler le mouvement et éventuellement des calques intermédiaires. On retrouvait aussi cette technique au cinéma où certains arrières-plans étaient peints.

4.1. Perspective atmosphérique

Voyons comment la technique des calques permet de simuler un paysage.
La ***perspective atmosphérique*** est une façon de représenter l'éloignement des différents plans ou calques par des teintes plus claires, des contrastes moins élevés et des couleurs plus grises (la saturation diminue). Cela reflète des phénomènes physiques réels, certaines particules de l'atmosphère (molécules d'air, gouttelettes d'eau, fumées...) diffractent la lumière (particulièrement les longueurs d'ondes plus courtes comme le bleu).



Il existe un modèle de couleurs HSB (*Hue/Saturation/Value* : Teinte/Saturation/Luminosité) pour lequel il suffit donc de changer le paramètre de saturation.
Ci-dessous les couleurs obtenues pour (h, s, b) avec $h = 0.66$, $b = 0.8$ et la saturation s variant de 0 à 1.



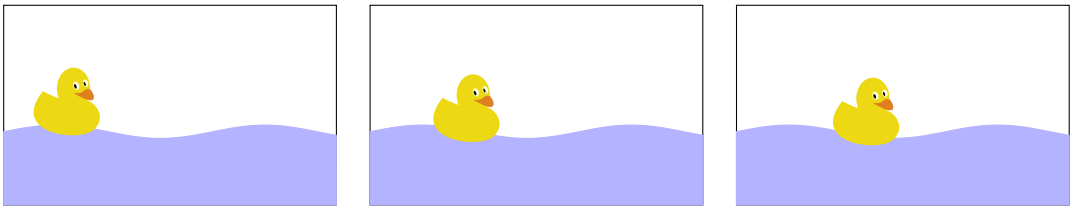
4.2. La 2.5D

Comme son nom le suggère, la 2.5D est un ensemble des techniques d'animation qui permettent de simuler une scène 3D. C'est le cas de beaucoup de jeux vidéos de stratégie où le point de vue du joueur est au-dessus du plateau de jeu. Les objets représentés en perspective (par exemple ci-dessous les immeubles) sont des petites images 2D pré-dessinées. C'est une technique efficace mais qui nécessite de fixer ou de limiter fortement le choix de la vue du joueur.

SimCity 2000 (images : Wikipedia)

4.3. Principe de l'animation 2D

Vous le savez, on simule le mouvement à l'écran par un succession rapide d'images. Ainsi les pixels ne se déplacent pas, mais seule leur couleur change. Le cerveau humain, entraîné par les objets réels en mouvement, reconstitue un déplacement à partir d'images fixes.



Pour réaliser de petits jeux animés, les modules de *Pygame* de *Python* sont très sympas. On explique ci-dessous les principes de l'animation 2D en suivant la philosophie de *Pygame*.

Animation 1D.

Commençons par une animation très élémentaire en une seule dimension. Les objets en jeu sont :

- `fond` qui est l'image d'arrière-plan représentée ici par une liste de `0`,

- un entier p qui représente la position d'un personnage, ce personnage est représenté par un `1`,
- `ecran` qui est l'image finale affichée, et sera donc une juxtaposition de `0` et de `1`.

Les étapes de l'animation sont les suivantes :

- *Initialisation.* On initialise `ecran` avec l'image d'arrière-plan `fond` :

  `0000000000`

  On définit une position de départ, par exemple $p = 3$.
- *Affichage.* Par `ecran`$[p] = 1$ on affiche le personnage :

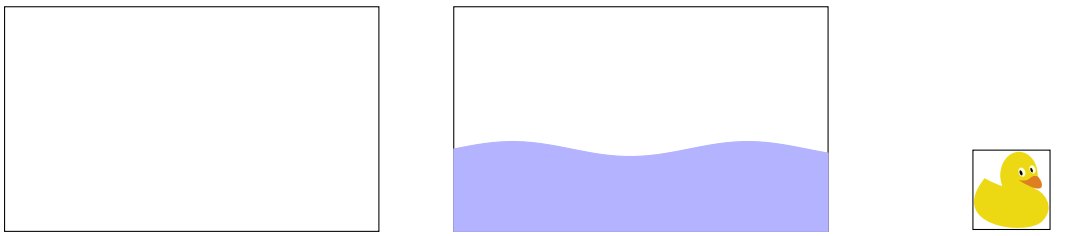
  `0001000000`
- *Déplacement.*
  - *Effacer le personnage.* `ecran`$[p] = 0$
  - *Changer la position.* $p = p + 1$.
  - *Réafficher le personnage.* `ecran`$[p] = 1$

En répétant le déplacement on obtient, successivement :

`0001000000`
`0000100000`
`0000010000`
`0000001000`
`0000000100`

Animation 2D.

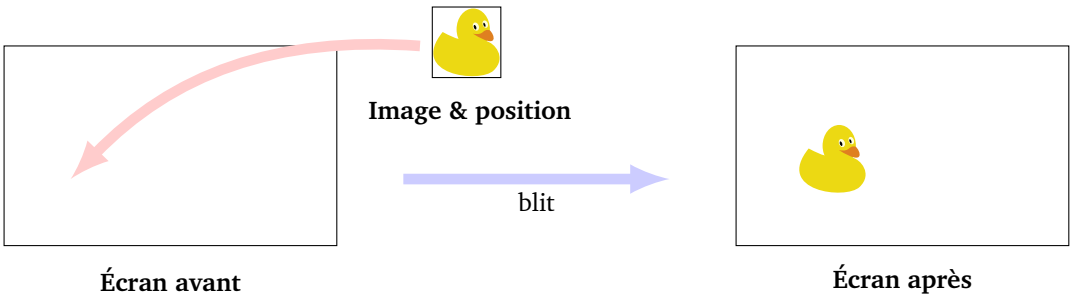
Le principe est similaire, il nous faut une image de fond, une image à afficher pour l'objet à déplacer. L'affichage à l'écran correspond à affecter des valeurs à la mémoire graphique (*screen buffer*) ; on représente ici l'écran comme une image.



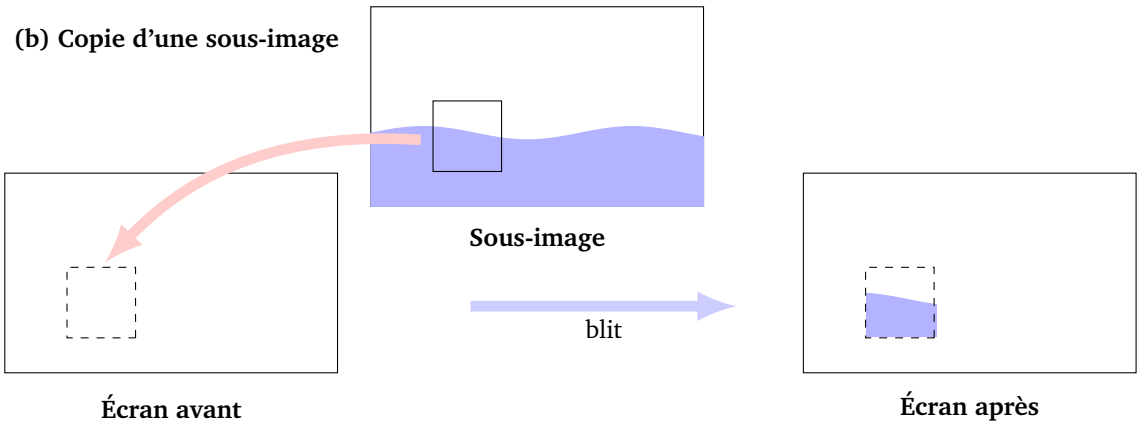
Écran **Arrière-plan** **Personnage**

L'opération fondamentale est la *copie rapide* ou *blit* (pour *bit-block transfer*) qui consiste à copier les données d'une image dans la mémoire graphique. Concrètement cela signifie copier une image et l'afficher à l'écran à une position donnée (figure (a)) ou bien copier seulement une partie d'une image (figure (b)).

(a) Copie d'une image



Écran avant **Écran après**



Le second cas de figure nous sert à effacer le personnage avant de le déplacer, plus exactement on copie à l'écran la portion de l'arrière-plan à la place qu'occupait le personnage.
Le processus de l'animation est alors le même que précédemment :

- *(a) Initialisation.* On initialise `ecran` avec l'image d'arrière-plan `fond` :
- *(b) Affichage.* On affiche le personnage à une position donnée.
- *Déplacement.* (À répéter)
  - *(c) Effacement du personnage.* Pour cela on copie la portion de l'arrière-plan correspondant à l'emplacement qu'occupe le personnage.
  - *(d) Affichage du personnage à sa nouvelle position.*

La vitesse du mouvement est contrôlée par la longueur du déplacement et la durée de la pause entre deux répétitions.