

Informatique Graphique

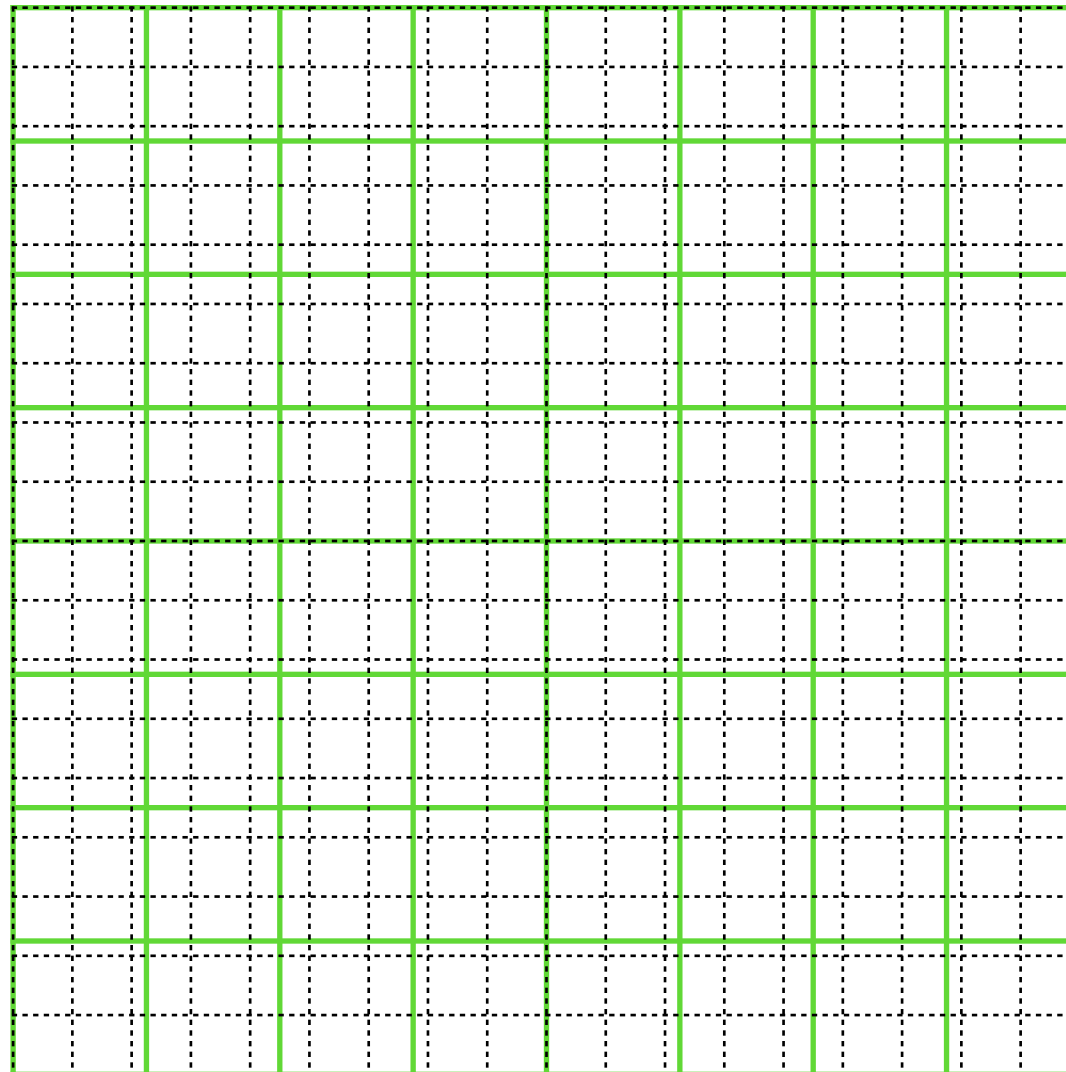
Agrandir/Réduire

Jean-Baptiste.Yunes@univ-paris-diderot.fr

9/2018

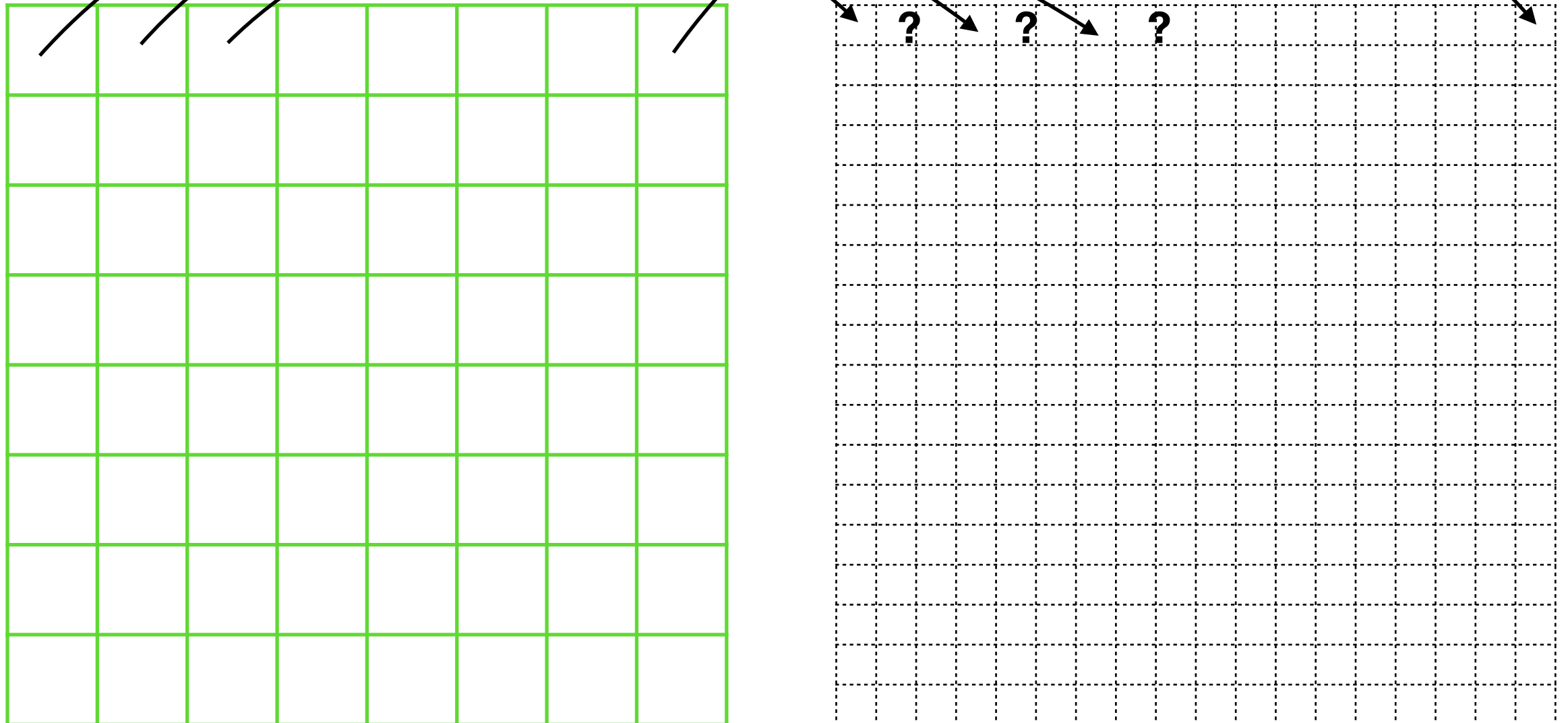
- Comment agrandir (zoom) ou rapetisser (unzoom) une image ?
- La difficulté est évidemment de conserver un aspect raisonnable...
 - Visuellement équivalent pour un humain
 - Informationnellement équivalent pour des algorithmes
- On peut aussi parler de mise à l'échelle (scaling)

- D'un espace à l'autre...



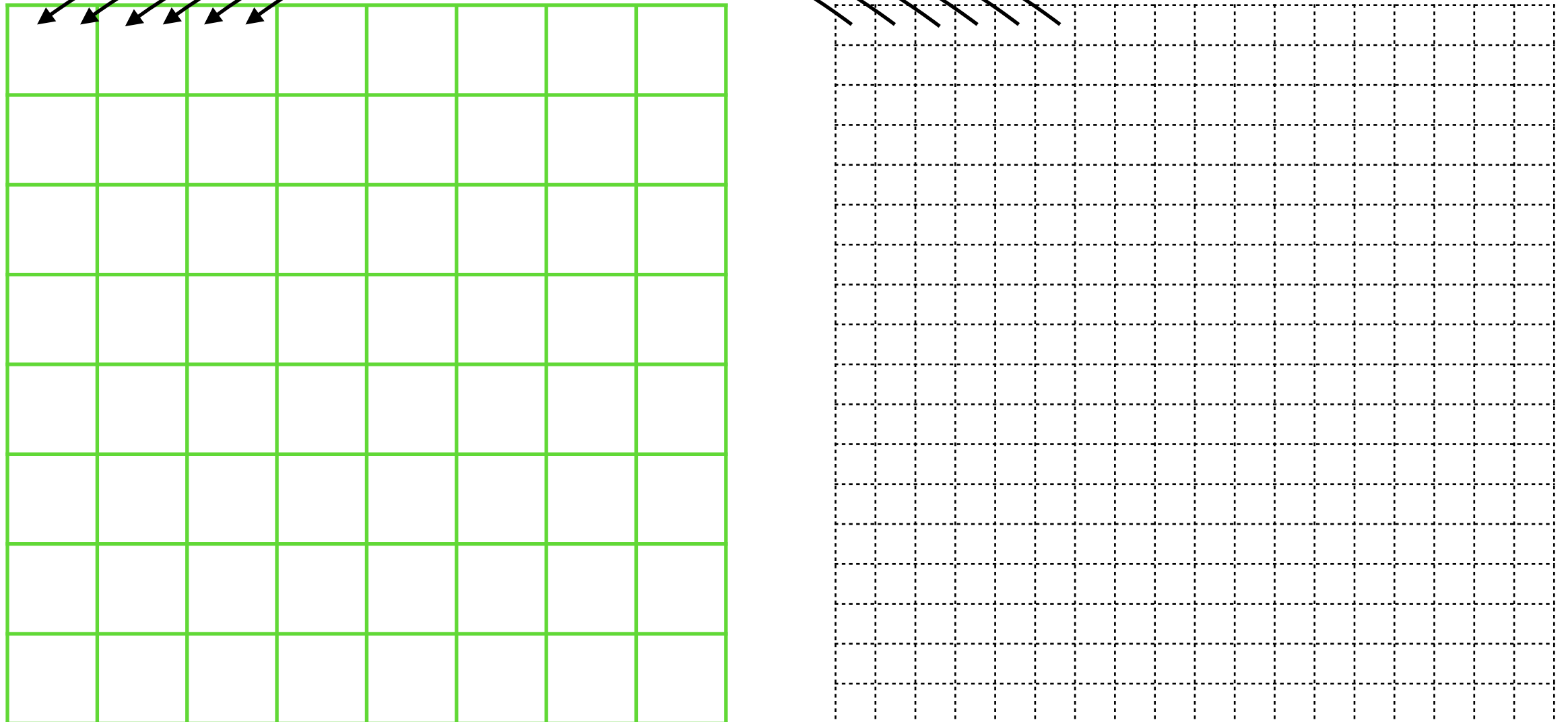
- La projection avant...

- des trous ?



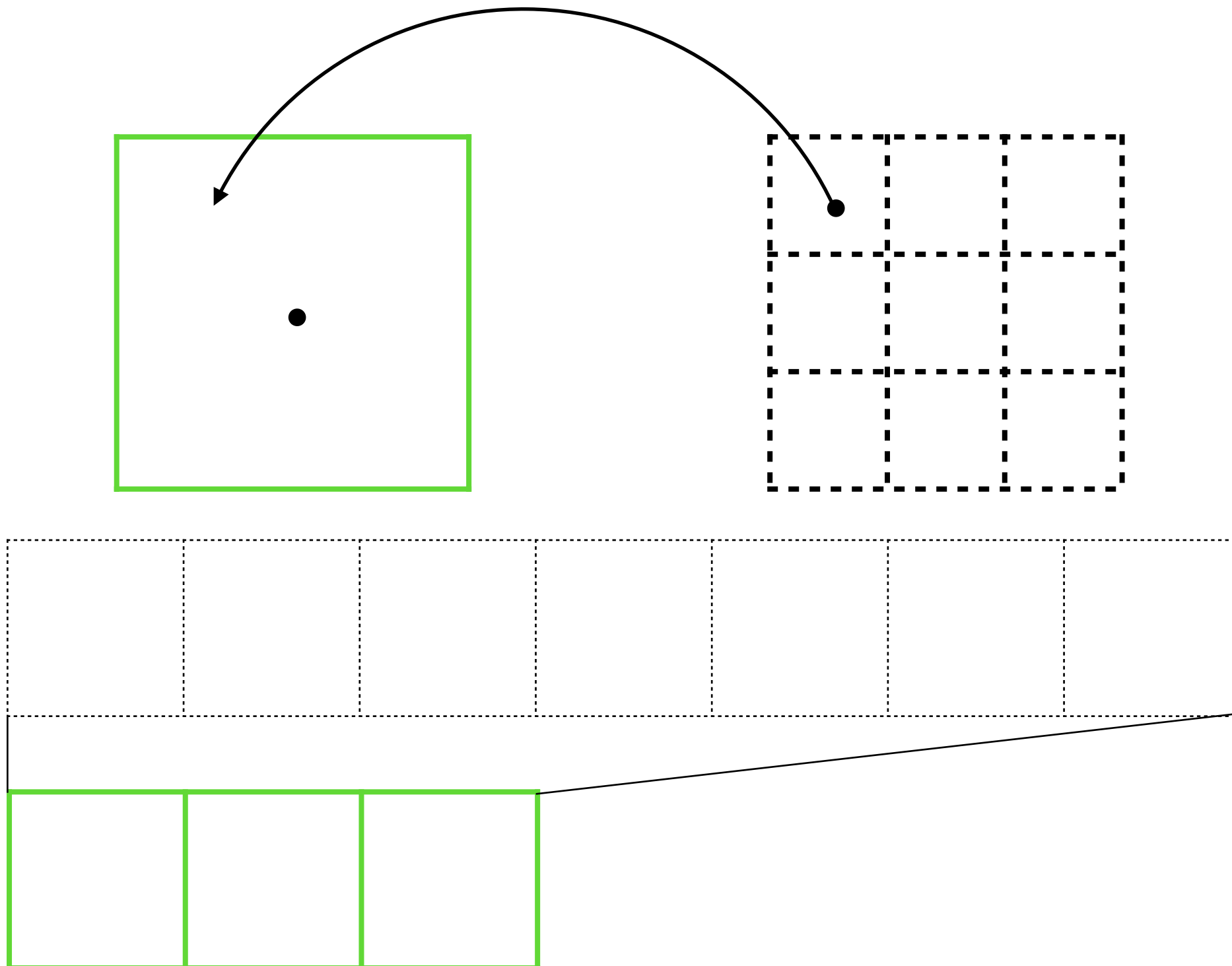
- La projection inverse...

- évite les «trous»

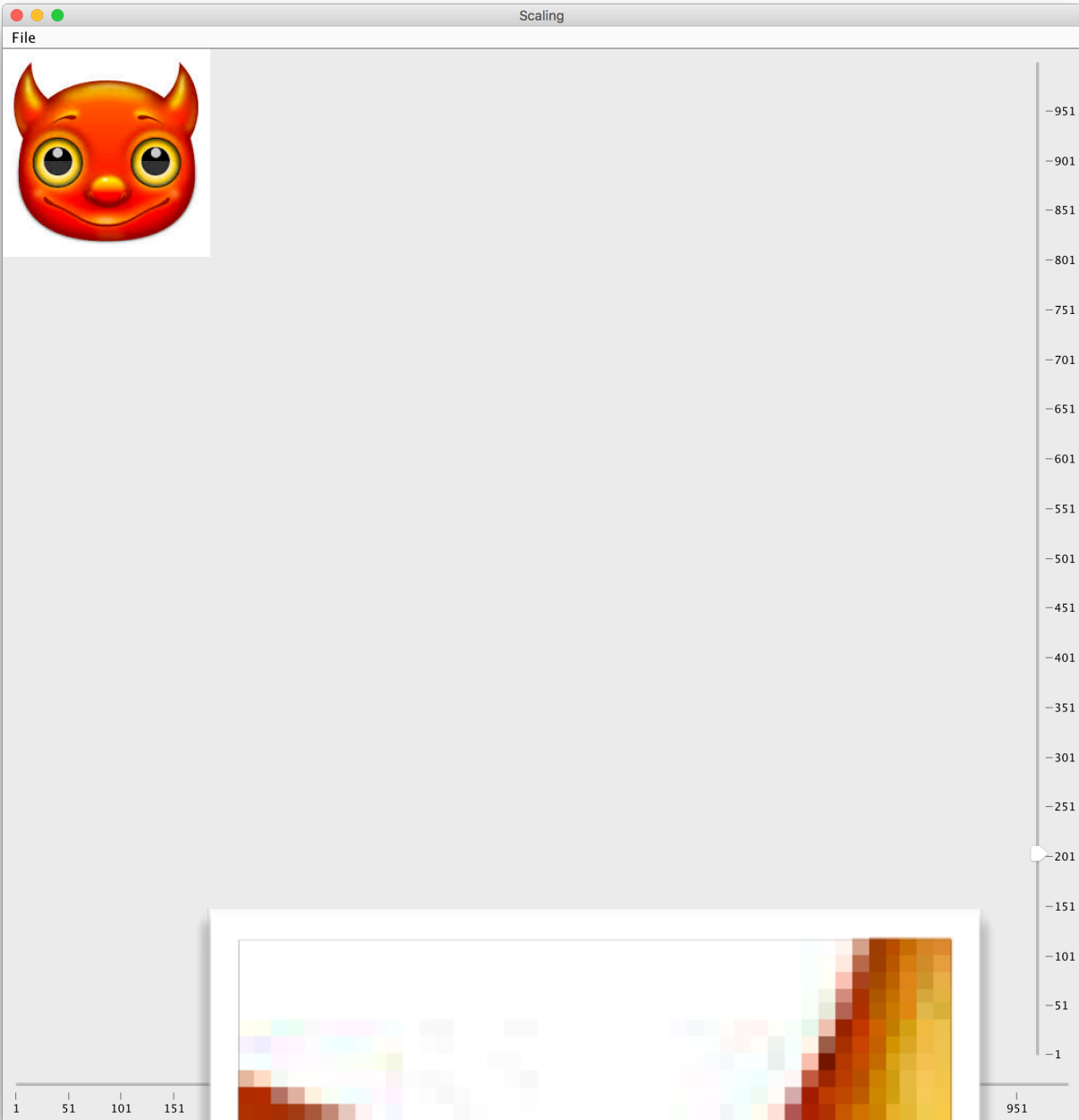


- Il n'y a pas de grossissement/réduction
 - Il y a des algorithmes qui permettent d'obtenir des effets de grossissement/réduction
 - Chacun a des propriétés différentes
 - Les résultats sont différents, mais aucun n'est juste ou faux
 - Le choix de l'algorithme doit être fait en conscience (c'est l'expertise qui dicte le choix)

- Plus proche voisin (Nearest Neighbor)



- $[W_d, H_d] \rightarrow [W_s, H_s]$
- `getDestinationPixelNN(xd, yd) {`
 $x_{\text{slope}} = W_s / W_d$
 $y_{\text{slope}} = H_s / H_d$
 `return getSourcePixel(xd*xslope, yd*yslope)`
}

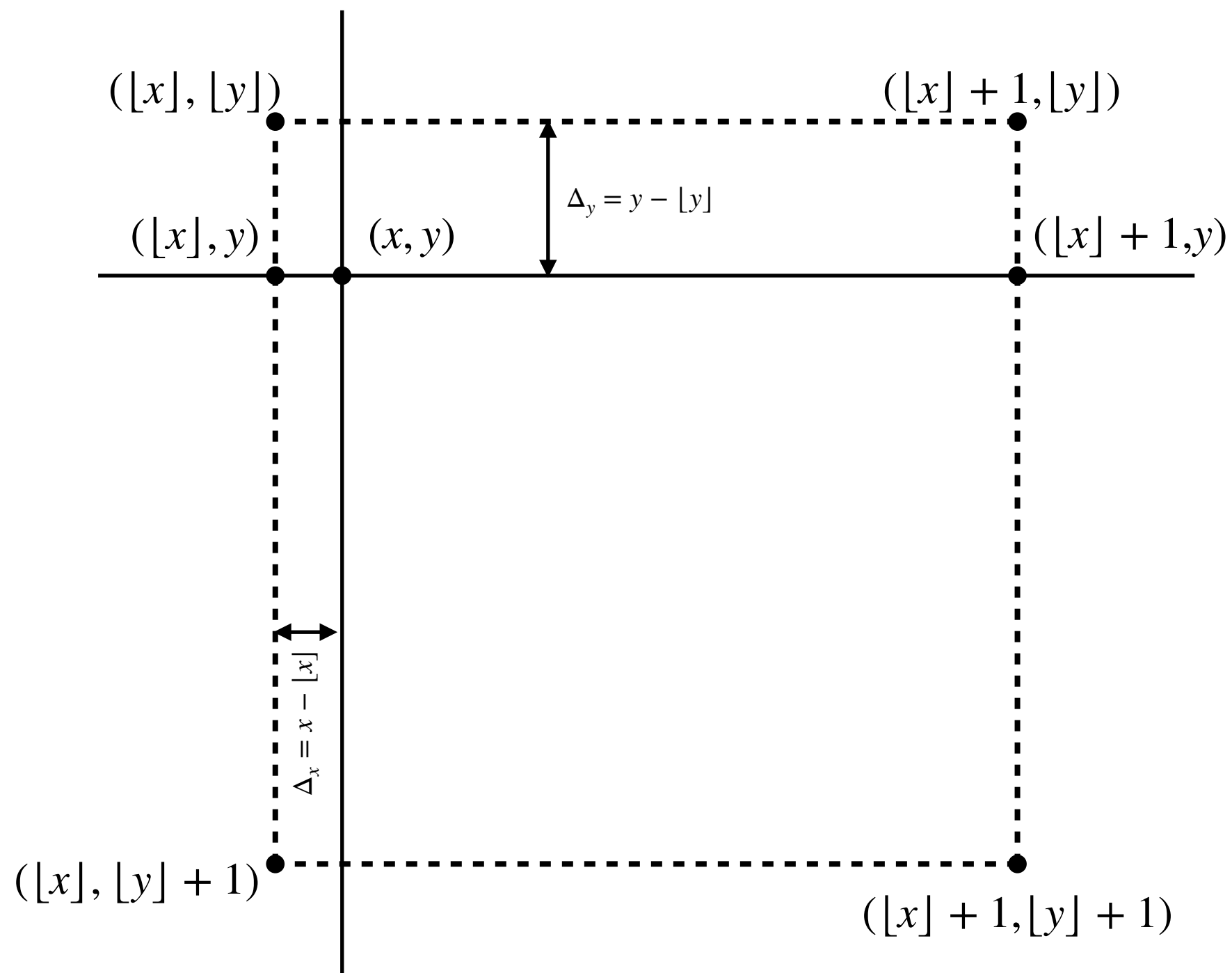


«Pixelisation» (Jaggedness)



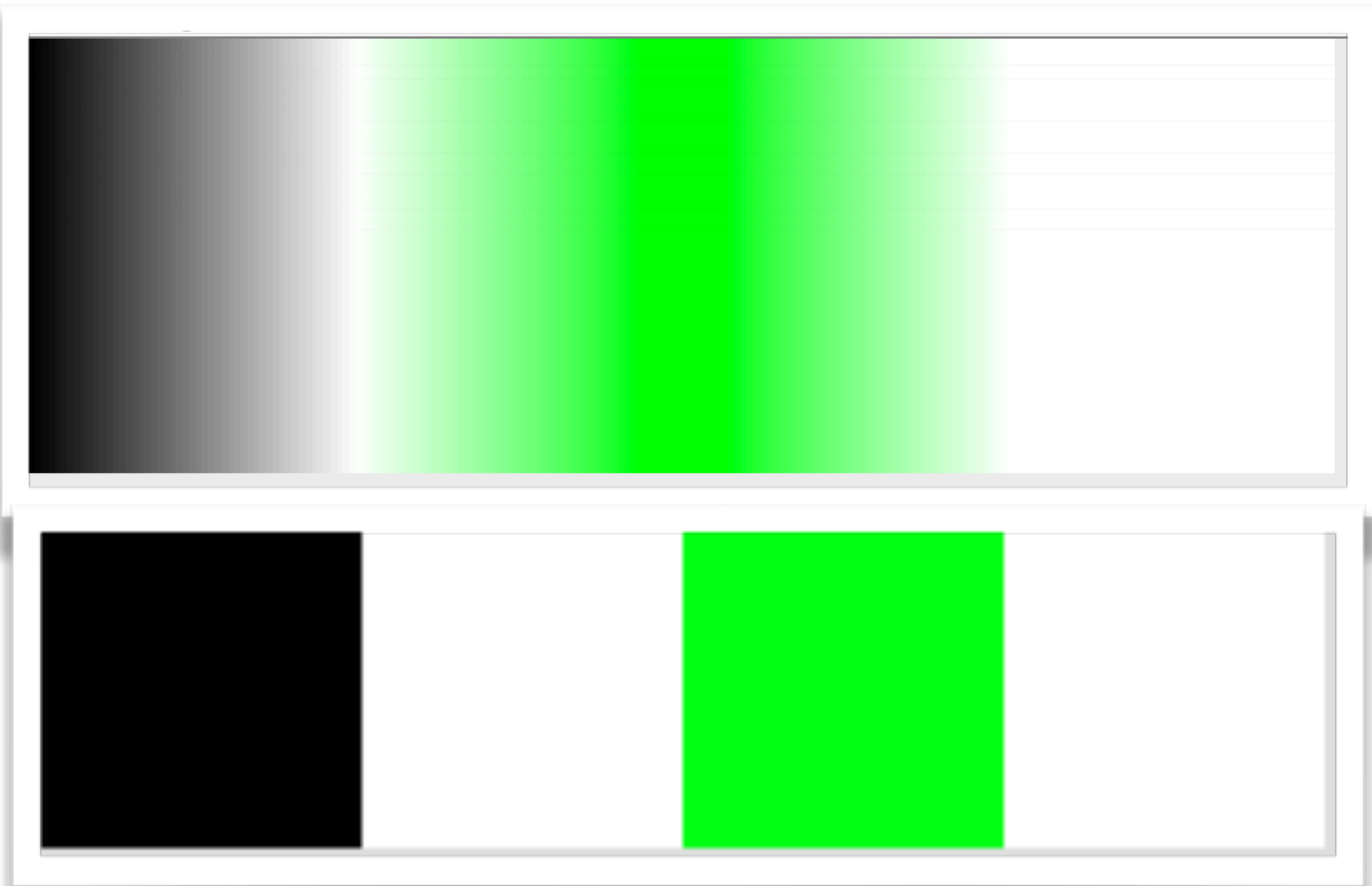
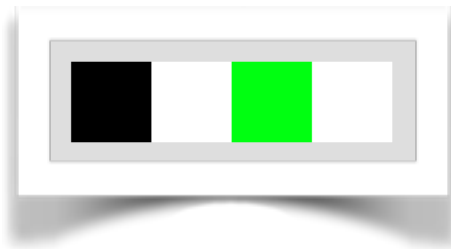
- Avantages
 - Rapide = recopie
- Inconvénients
 - Destructif
 - Grossissement : Pixelisation (Jaggedness)
 - Réduction : Perte d'information
- Lien avec Bresenham ?
 - Anti-crénelage ???

- Approximation linéaire...
 - La projection n'est pas exacte
 - NN arrondi simplement
 - l'approximation linéaire est obtenu par une moyenne pondérée
- Elle agit dans les deux dimensions à la fois d'où le nom
 - Bi-linéaire (Bi-linear)



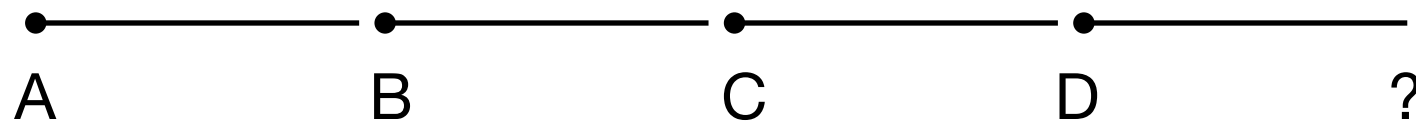
- Un point sur le segment A,B est $(1-t)A+tB$ pour tout t dans $[0,1]$
- $$v(x, y) = (1 - \Delta_y) \left[(1 - \Delta_x) v(x_i, y_i) + \Delta_x v(x_i + 1, y_i) \right] + \Delta_y \left[(1 - \Delta_x) v(x_i, y_i + 1) + \Delta_x v(x_i + 1, y_i + 1) \right]$$

- $[W_d, H_d] \rightarrow [W_s, H_s]$
- `getDestinationPixel(xd, yd) {`
 `double xslope = Ws / Wd`
 `double yslope = Hs / Hd`
 `xi = (int) (xd+0.5)*xslope`
 `yi = (int) (yd+0.5)*yslope`
 `double dx = (xd+0.5)*xslope - xi`
 `double dy = (yd+0.5)*yslope - yi`
 `v00 = getSourcePixel(xi, yi)`
 `v10 = getSourcePixel(xi+1, yi)`
 `v01 = getSourcePixel(xi, yi+1)`
 `v11 = getSourcePixel(xi+1, yi+1)`
 `return (1-dy)((1-dx)v00+dxv10) + dy((1-dx)v01+dxv11)`
 `}`
- Attention aux bornes... $x_{i+1}=W_s$ et $y_{i+1}=H_s$

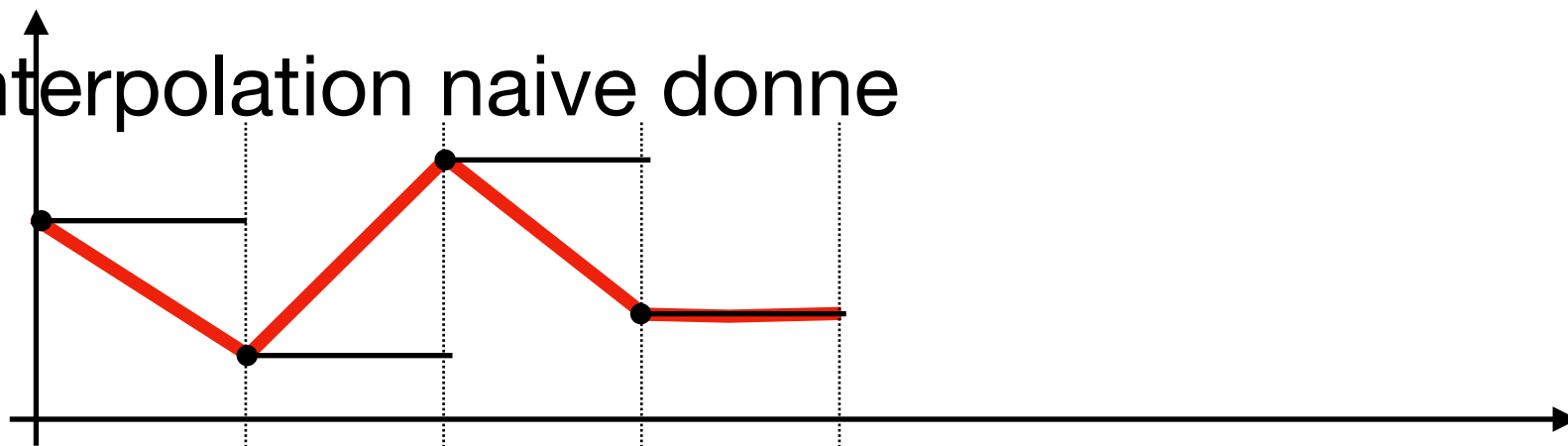


- Problème...
- Attention le facteur de grossissement est énorme (environ 300x, l'original étant de 4x1)
- On a donc dû inventer de très(trop) nombreuses valeurs
- Conditions aux bords...

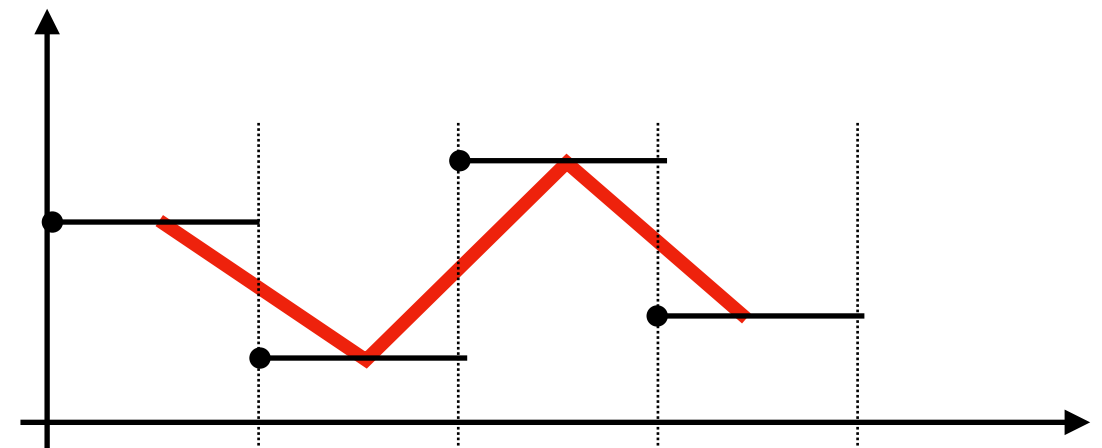
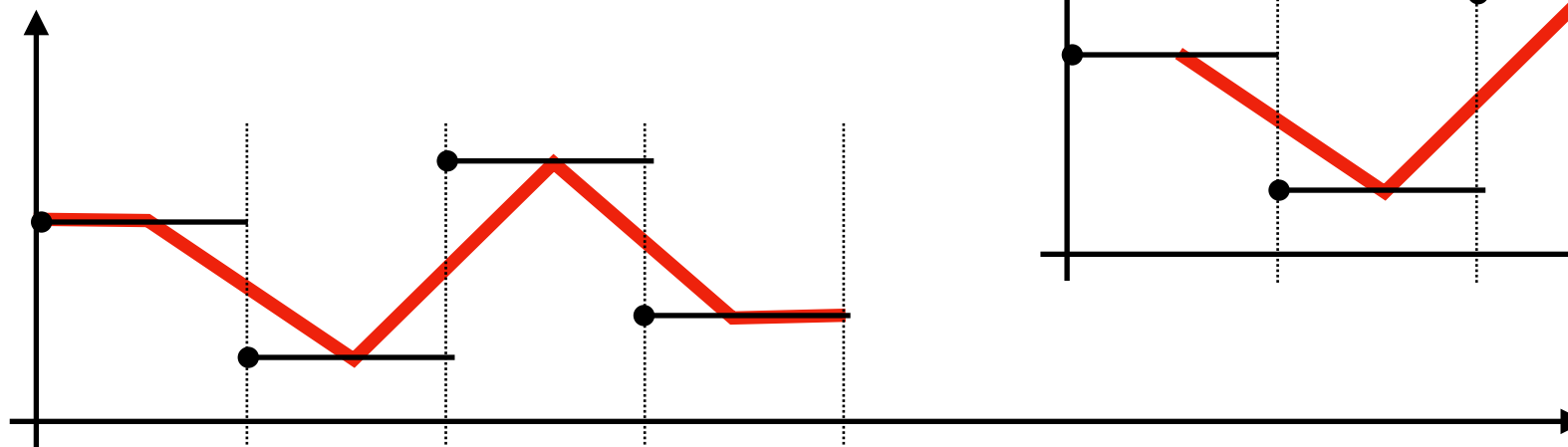
- Le problème est l'introduction d'un biais au bord...
- Les pixels sont des semi-ouverts



- donc l'interpolation naïve donne

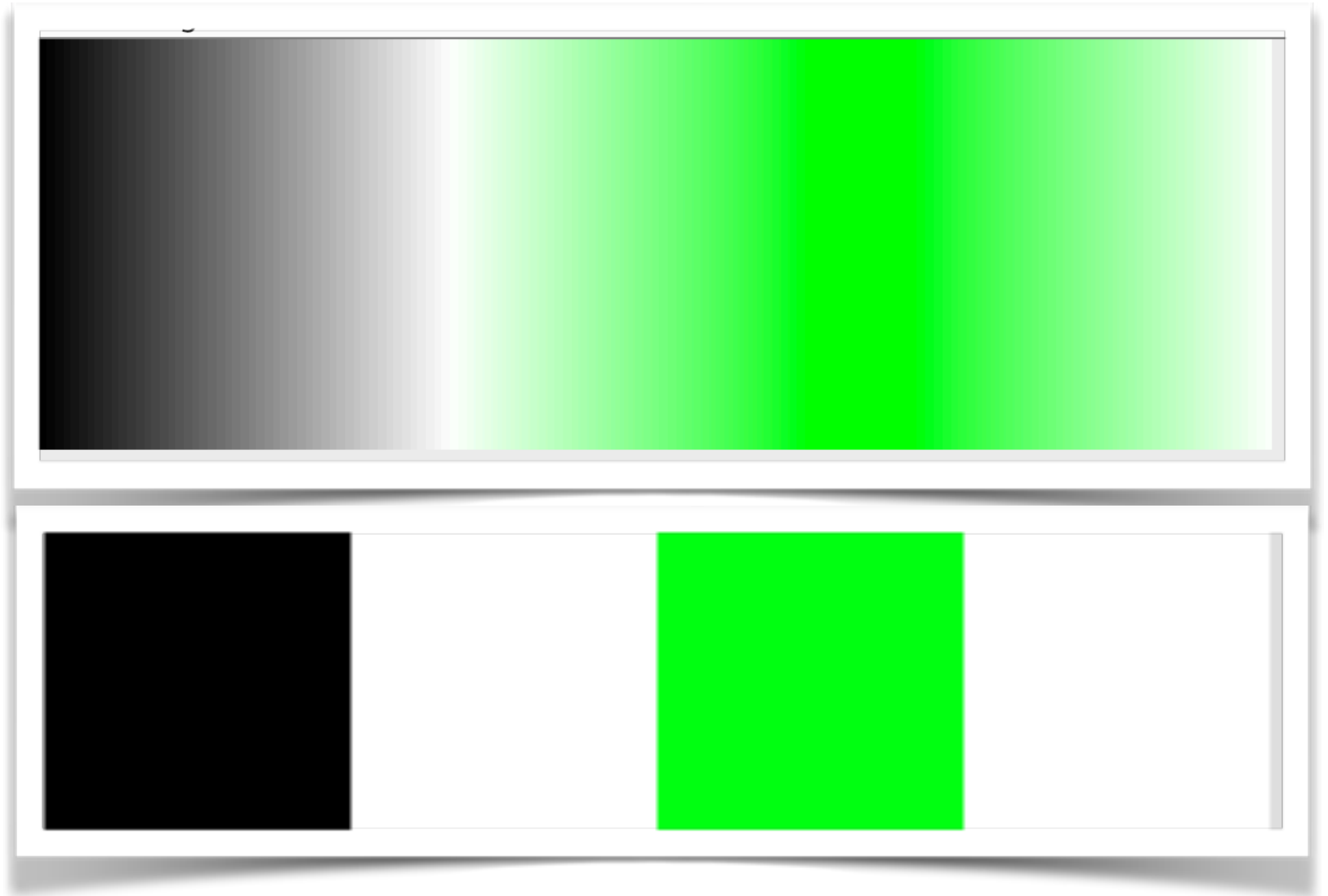
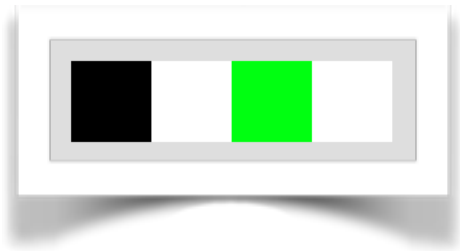


- on préfère ?



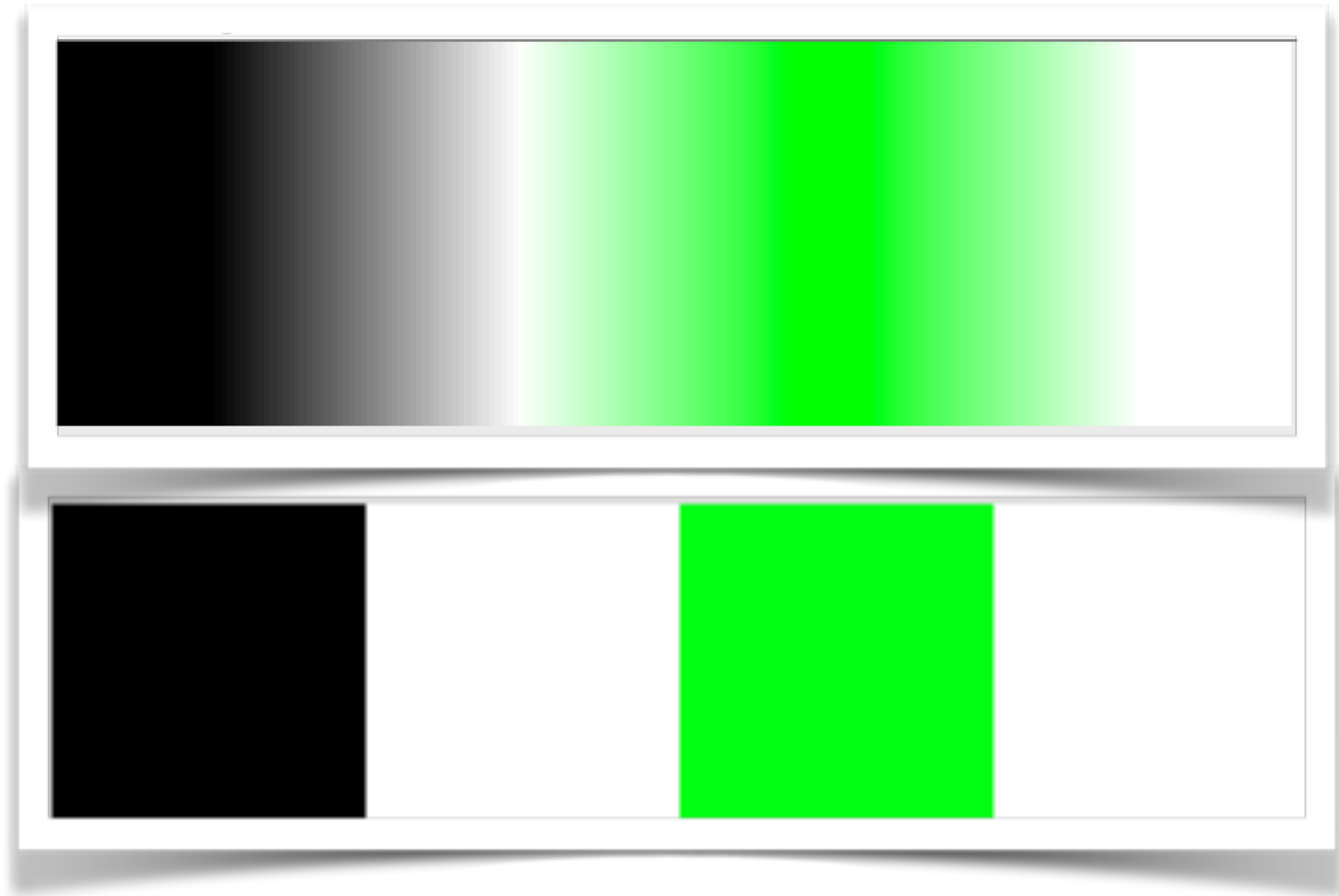
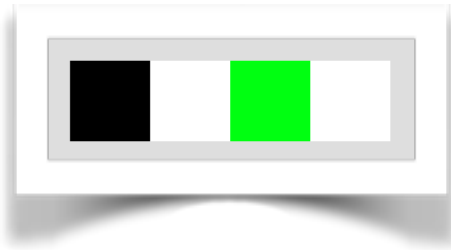
- `getDestinationPixel(xd,yd) {`
 $x_{\text{slope}} = (W_s - 1) / W_d$
 $y_{\text{slope}} = (H_s - 1) / H_d$
 $x_s = (x_d + 0.5) * x_{\text{slope}}$
 $y_s = (y_d + 0.5) * y_{\text{slope}}$
 $x_i = \text{(int)}x_s, x_{i+1} = x_i + 1$
 $y_i = \text{(int)}y_s, y_{i+1} = y_i + 1$
 $d_x = x_s - x_i$
 $d_y = y_s - y_i$
 $v_{00} = \text{getSourcePixel}(x_i, y_i)$
 $v_{10} = \text{getSourcePixel}(x_{i+1}, y_i)$
 $v_{01} = \text{getSourcePixel}(x_i, y_{i+1})$
 $v_{11} = \text{getSourcePixel}(x_{i+1}, y_{i+1})$
 $\text{return } (1 - d_y)((1 - d_x)v_{00} + d_x v_{10}) + d_y((1 - d_x)v_{01} + d_x v_{11})$
`}`

- Attention aux bornes x_{i+1} et y_{i+1}



- Résultat un peu étrange mais interprétable...

- `getDestinationPixel(xd,yd) {`
 $x_{\text{slope}} = (W_s - 1) / W_d, y_{\text{slope}} = (H_s - 1) / H_d$
 $x_s = (x_d + 0.5) * x_{\text{slope}}, y_s = (y_d + 0.5) * y_{\text{slope}}$
`if (xs - (int)xs < 0.5) xi = (int)xs - 1 else xi = (int)xs;`
`if (ys - (int)ys < 0.5) yi = (int)ys - 1 else yi = (int)ys;`
 $x_{i+1} = x_i + 1$
 $y_{i+1} = y_i + 1$
`if (xi < 0) xi = 0; if (xi+1 >= Ws) xi+1 = Ws - 1`
`if (yi < 0) yi = 0; if (yi+1 >= Hs) yi+1 = Hs - 1`
 $v_{00} = \text{getSourcePixel}(x_i, y_i)$
 $v_{10} = \text{getSourcePixel}(x_{i+1}, y_i)$
 $v_{01} = \text{getSourcePixel}(x_i, y_{i+1})$
 $v_{11} = \text{getSourcePixel}(x_{i+1}, y_{i+1})$
`if (xs - (int)xs < 0.5) dx = xs - xi + 0.5 else dx = xs - xi - 0.5`
`if (ys - (int)ys < 0.5) dy = ys - yi + 0.5 else dy = ys - yi - 0.5`
`return (1 - dy)((1 - dx)v00 + dxv10) + dy((1 - dx)v01 + dxv11)`
`}`



- L'interprétation visuelle est tout à fait raisonnable!
- Si on seuille...

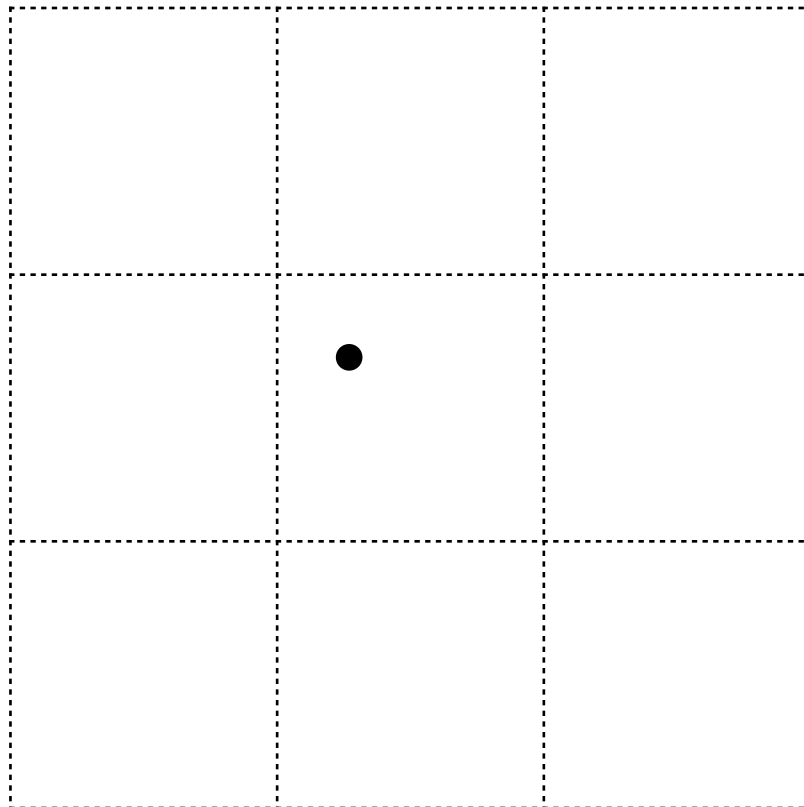
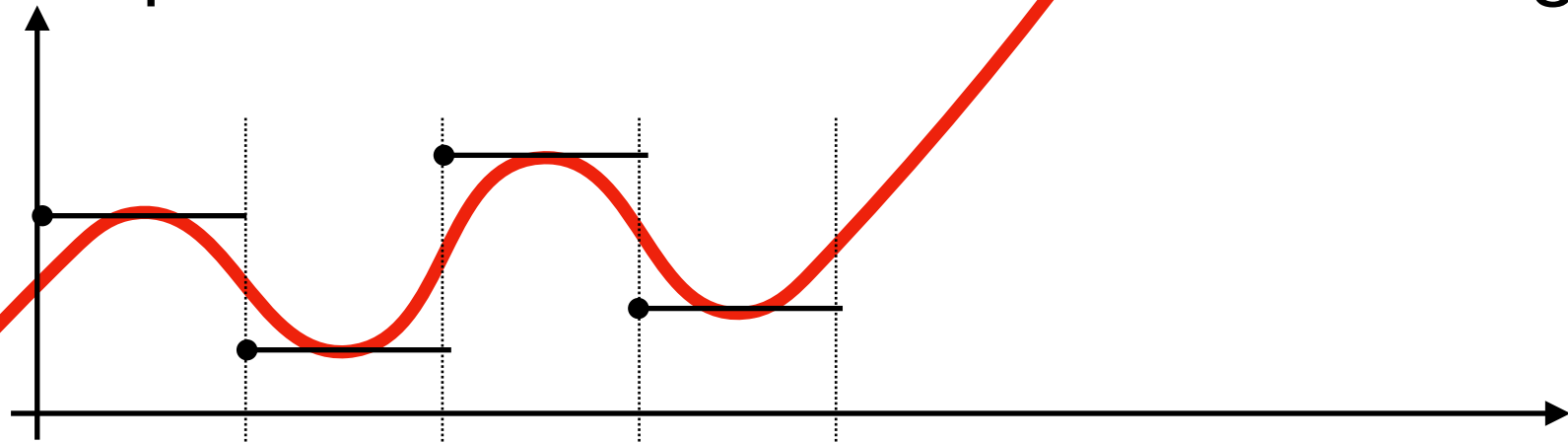


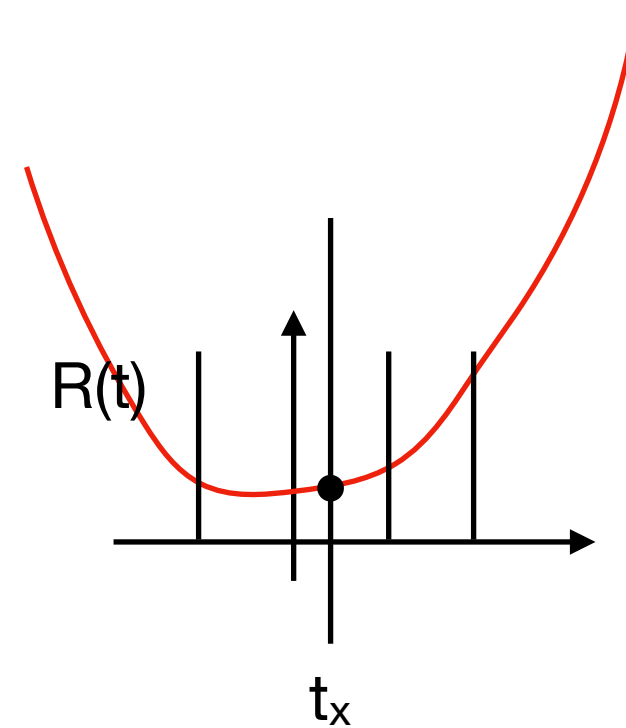
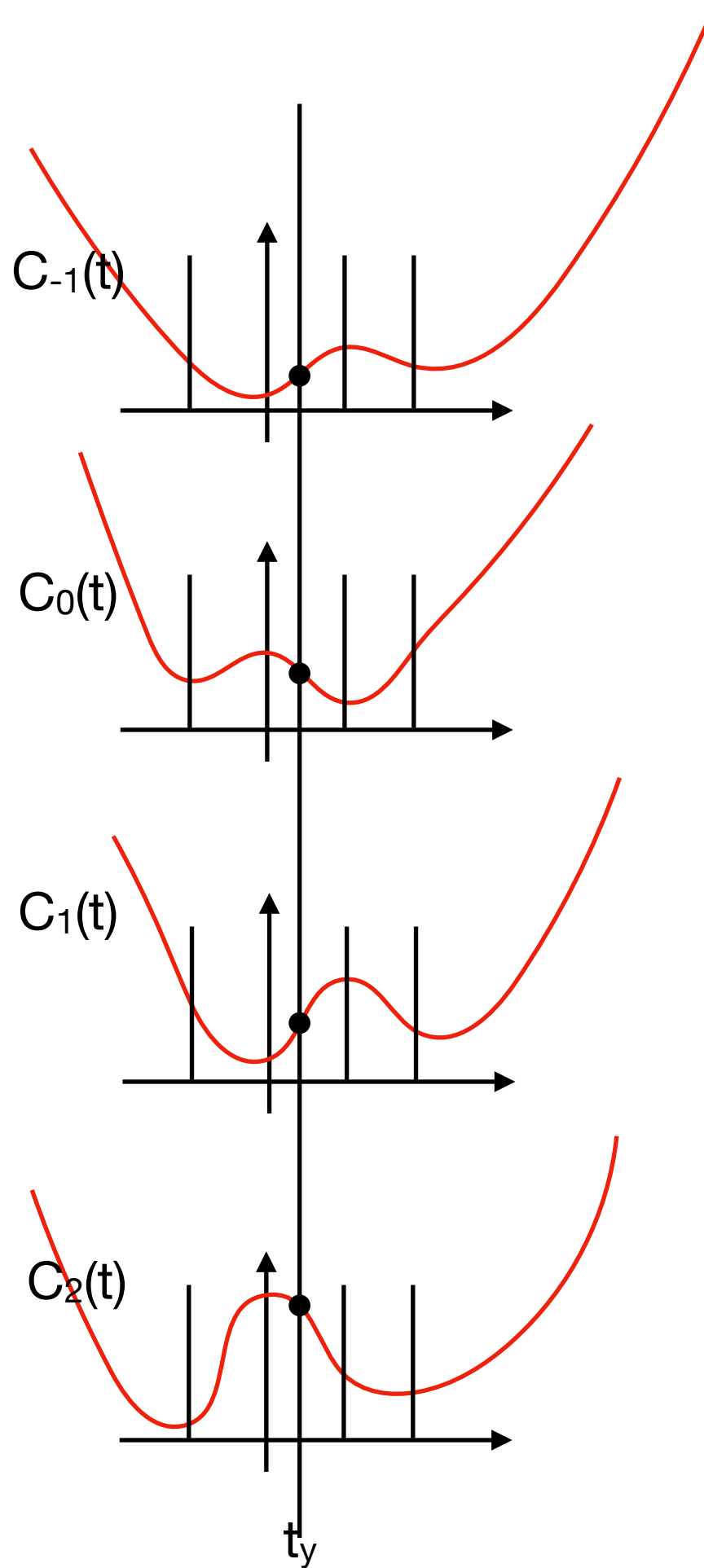
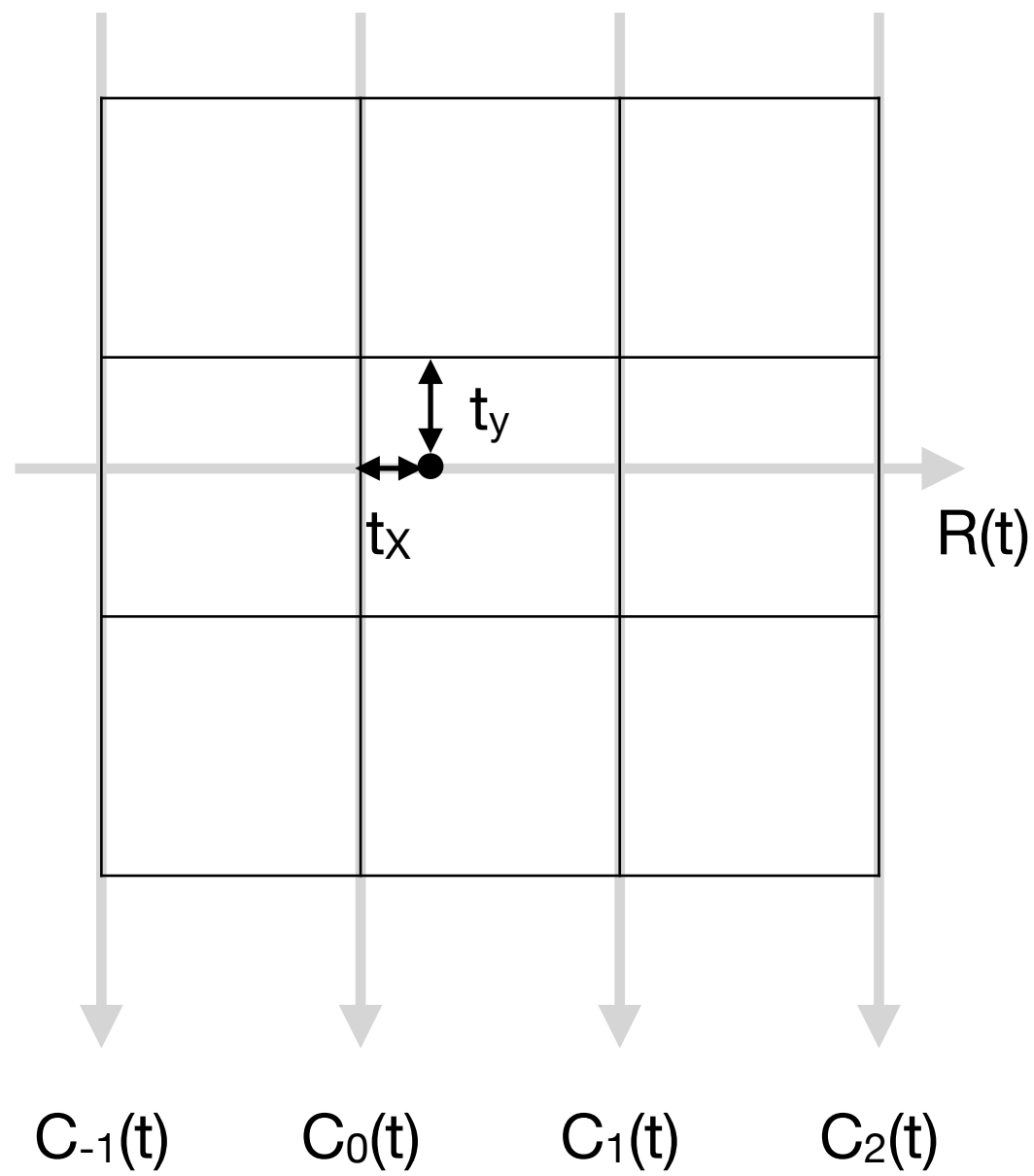
Nearest neighbor



- Avantages
 - Résultat généralement plus satisfaisant que NN lorsqu'il s'agit d'une «photographie»
- Inconvénients
 - Réduit les contraintes
 - Coût algorithmique un peu plus élevé, mais reste très raisonnable

- Bi-cubique
- en 4 points passe une et une seule courbe de degré 3





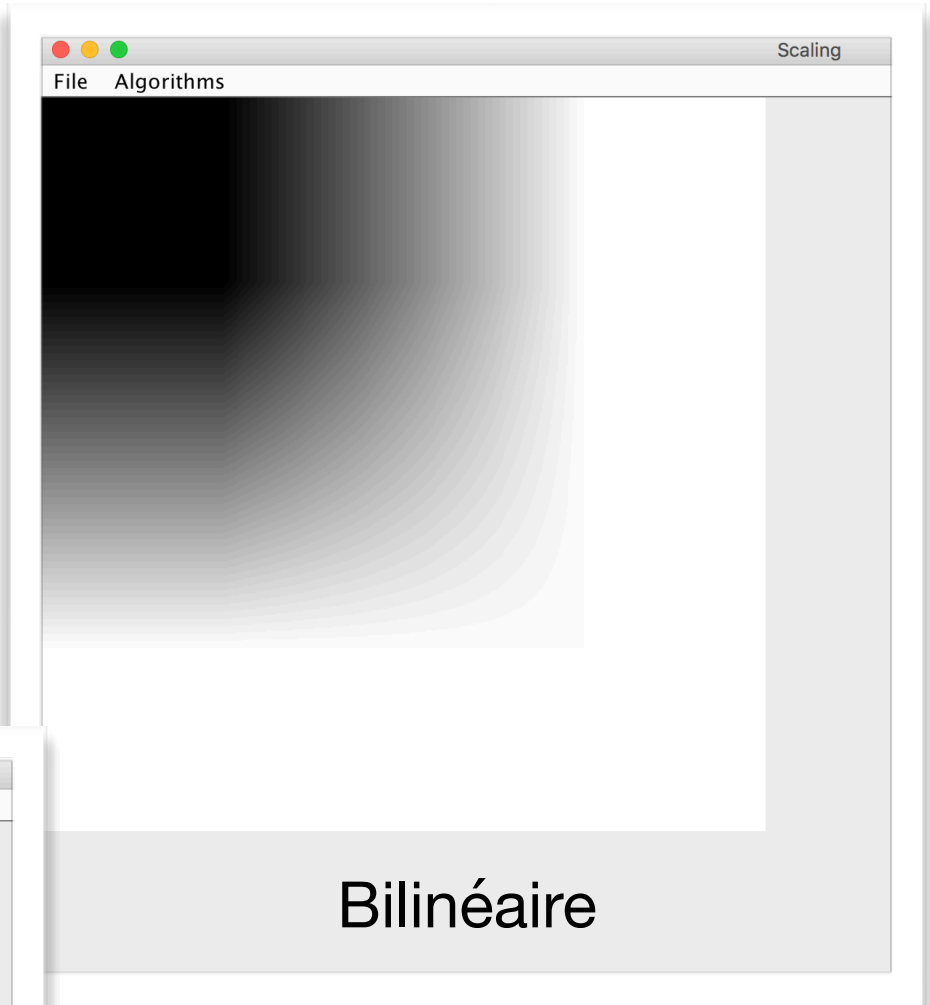
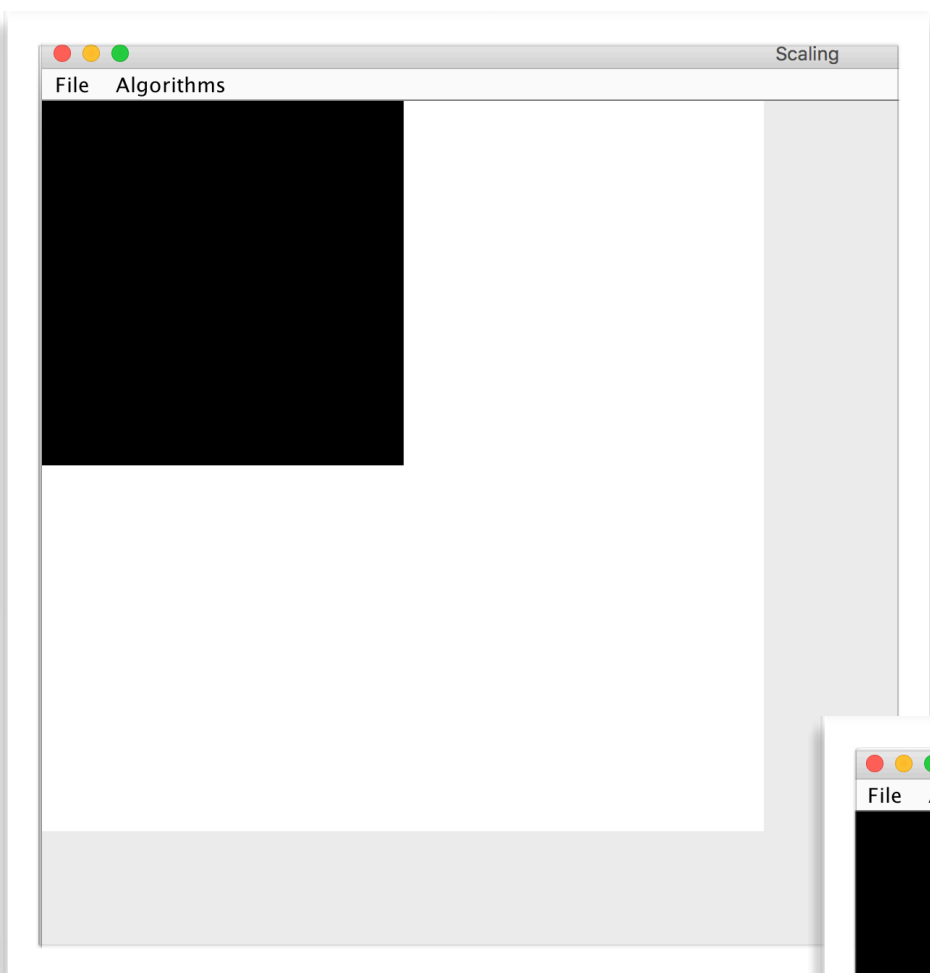
- Polynôme de degré 3 : $P(t) = at^3 + bt^2 + ct + d$

$$\begin{cases} P(-1) &= & -a + b - c + d \\ P(0) &= & d \\ P(1) &= & a + b + c + d \\ P(2) &= & 8a + 4b + 2c + d \end{cases}$$

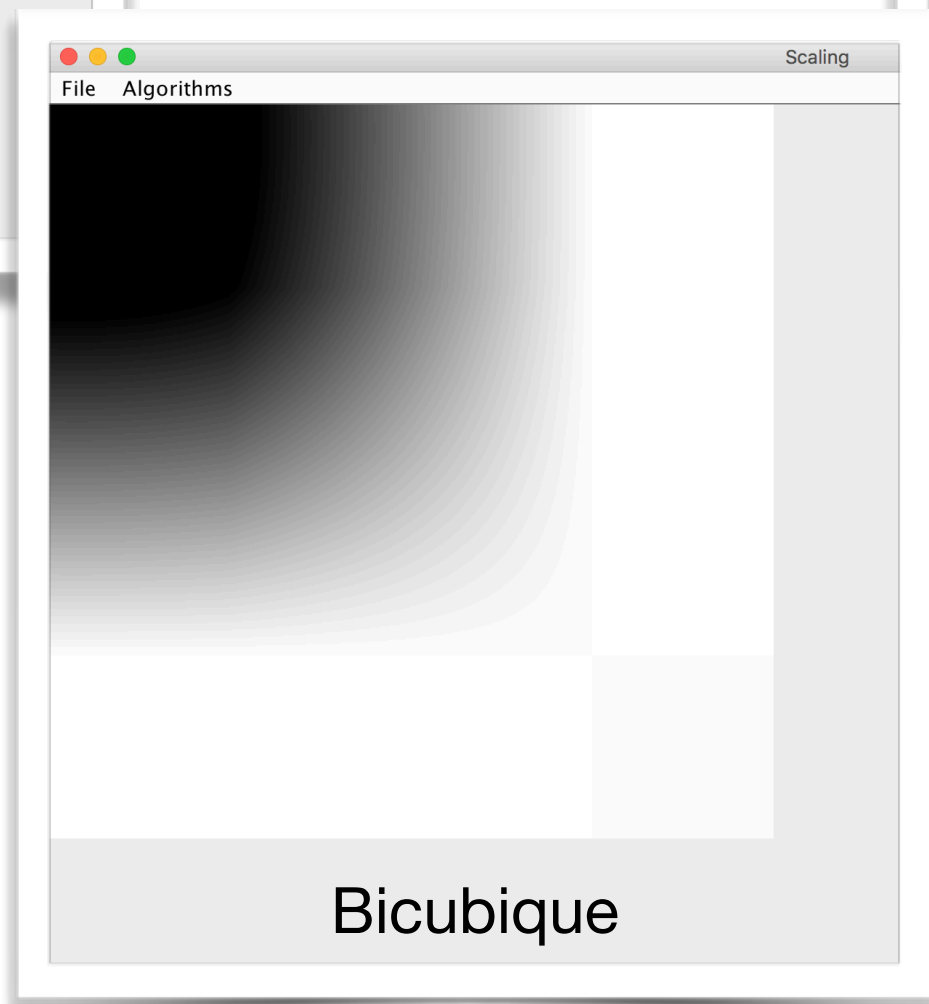
- par inversion de matrice

$$\begin{cases} a &= & -\frac{1}{6}P(-1) & +\frac{1}{2}P(0) & -\frac{1}{2}P(1) & +\frac{1}{6}P(2) \\ b &= & \frac{1}{2}P(-1) & -P(0) & +\frac{1}{2}P(1) & \\ c &= & -\frac{1}{3}P(-1) & -\frac{1}{2}P(0) & +P(1) & -\frac{1}{6}P(2) \\ d &= & & P(0) & & \end{cases}$$

- `getDestinationPixel(xd,yd) {`
 $x_{\text{slope}} = (W_s - 1) / W_d, y_{\text{slope}} = (H_s - 1) / H_d$
 $x_s = (x_d + 0.5) * x_{\text{slope}}, y_s = (y_d + 0.5) * y_{\text{slope}}$
`if (xs - (int)xs < 0.5) xi = (int)xs - 1 else xi = (int)xs;`
`if (ys - (int)ys < 0.5) yi = (int)ys - 1 else yi = (int)ys;`
`xi-1 = xi - 1, yi-1 = yi - 1`
`xi+1 = xi + 1, yi+1 = yi + 1`
`xi+2 = xi+1 + 1, yi+1 = yi+1 + 1`
`if (xi-1 < 0) xi-1 = 0; if (yi-1 < 0) yi-1 = 0;`
`if (xi < 0) xi = 0; if (yi < 0) yi = 0;`
`if (xi+1 >= Ws) xi+1 = Ws - 1, if (yi+1 >= Hs) yi+1 = Hs - 1`
`if (xi+2 >= Ws) xi+2 = Ws - 1, if (yi+2 >= Hs) yi+2 = Hs - 1`
`py-1 = cubic(sourcePix(xi-1,yi-1), sourcePix(xi,yi-1),,sourcePix(xi+1,yi-1),sourcePix(xi+2,yi-1))`
`py = cubic(sourcePix(xi-1,yi), sourcePix(xi,yi),,sourcePix(xi+1,yi),sourcePix(xi+2,yi))`
`py+1 = cubic(sourcePix(xi-1,yi+1), sourcePix(xi,yi+1),,sourcePix(xi+1,yi+1),sourcePix(xi+2,yi+1))`
`py+2 = cubic(sourcePix(xi-1,yi+2), sourcePix(xi,yi+2),,sourcePix(xi+1,yi+2),sourcePix(xi+2,yi+2))`
`if (xs - (int)xs < 0.5) dx = xs - xi + 0.5 else dx = xs - xi - 0.5`
`if (ys - (int)ys < 0.5) dy = ys - yi + 0.5 else dy = ys - yi - 0.5`
`return cubic(py-1,py,py+1,py+2)`
`}`



Bilinéaire



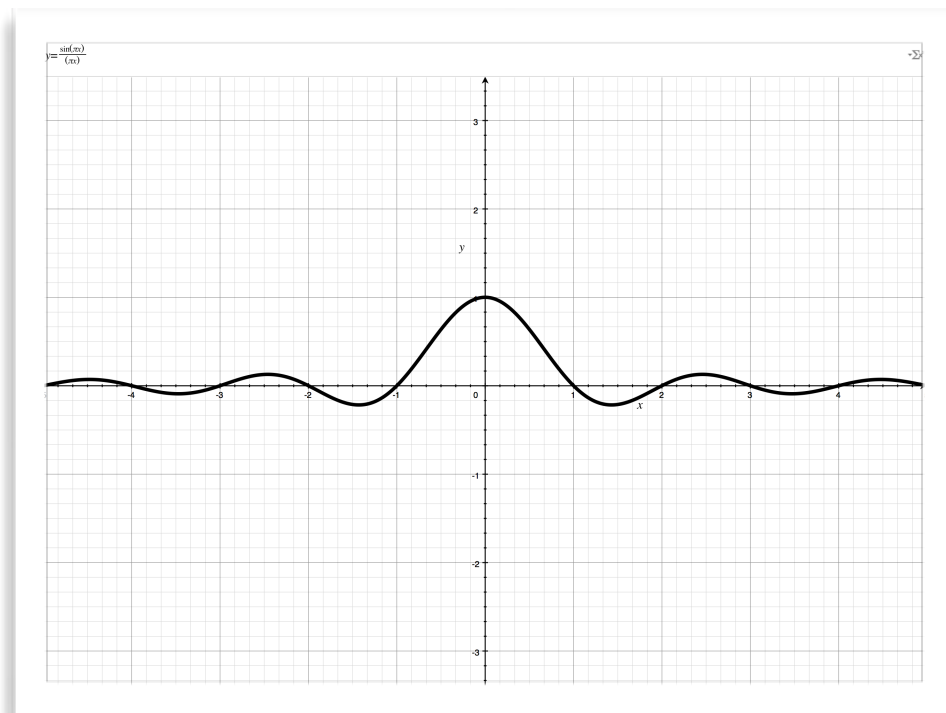
Bicubique

- Avantages
 - Moins de biais horizontaux et verticaux
 - Plus contrasté
- Inconvénients
 - Plus coûteux en calcul (sensiblement)

- Lanczos
 - Toujours la même idée
 - Reconstruire une fonction continue
- Note : bicubique est une approximation de Lanczos

- La fonction sinus cardinal est la fonction :

$$\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \text{ avec } \text{sinc}(0) = 1$$



- La fonction vaut 1 en 0, et 0 sur les autres entiers

- La fonction de Lanczos est :

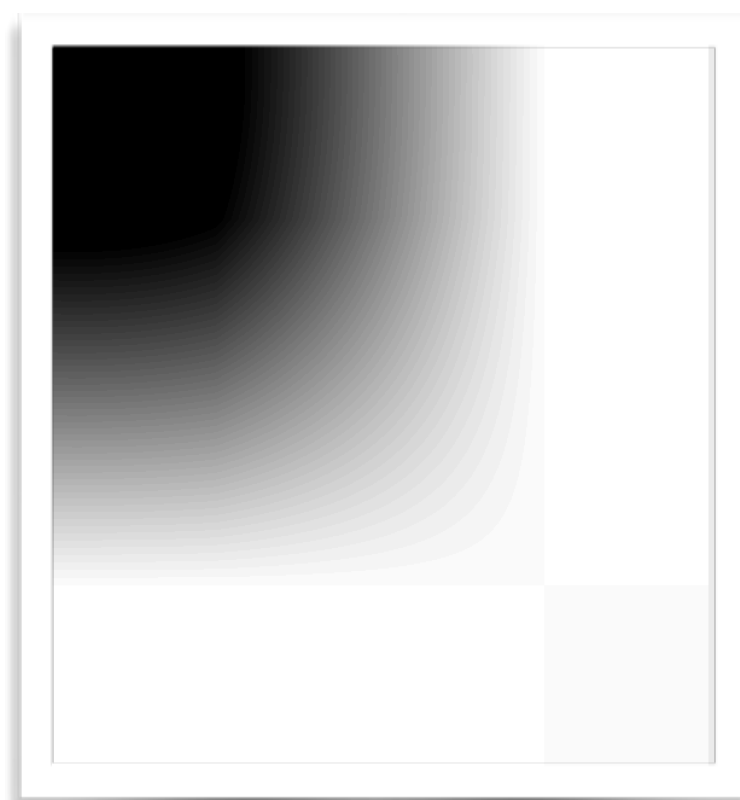
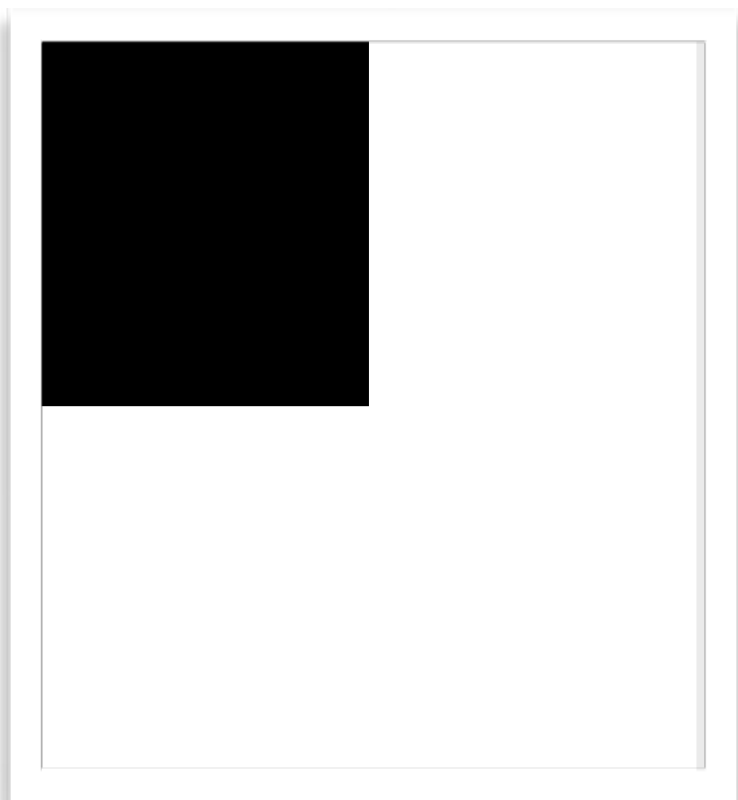
$$L(x) = \begin{cases} \text{sinc}(x) \text{sinc}(\frac{x}{a}) & \text{si } x \in [-a, a] \\ 0 & \text{sinon} \end{cases}$$

- L'interpolation est obtenue par convolution discrete avec les points :

$$S(x) = \sum_{i=-a}^a S_i L(x - i)$$

- aux points entiers on aura $S_i \dots$
- pour les points intermédiaires on aura une interpolation

- Solution (grossière)
- `getDestinationPixel(xd,yd) {`
 $x_{\text{slope}} = (W_s - 1) / W_d, y_{\text{slope}} = (H_s - 1) / H_d$
 $x_s = (x_d + 0.5) * x_{\text{slope}}, y_s = (y_d + 0.5) * y_{\text{slope}}$
 if $(x_s - \text{int}(x_s) < 0.5)$ $x_0 = \text{int}(x_s) - 1$ else $x_0 = \text{int}(x_s)$;
 if $(y_s - \text{int}(y_s) < 0.5)$ $y_0 = \text{int}(y_s) - 1$ else $y_0 = \text{int}(y_s)$;
 for j dans [-a,a]
 $p_j = \text{lanczos}(\{ (x_0 - i, y_0 - j), \text{ pour } i \text{ dans } [-a, a] \}, x_s - \text{int}(x_s))$
 return $\text{lanczos}(\{ p_j \text{ pour } j \text{ dans } [-a, a] \}, y_s - \text{int}(y_s))$
`}`



- Avantages
 - Une meilleure reconstruction
 - Conserve mieux les détails et marque mieux les bords
- Inconvénients
 - Introduit des artefacts le long des bords
 - Très coûteux algorithmiquement

- Ces algorithmes sont tous les mêmes (leur squelette) :
 - prendre un voisinage du point x_s, y_s (le projeté inverse)
 - lui appliquer une fonction bi-dimensionnelle du voisinage en $t_x = x_s - x_0$, $t_y = y_s - y_0$