TP 2 Traitement d’image - Hough

ISMAIL Amine

GIRARD Thierry

ISMAIL Amine – GIRARD Thierry

Exercice 1

Question 1

On aura :

- 100/2 = 50 : [1;3;5;...;100]
- 100/0,5 = 200 : [1;1.5;...;100] => Coordonnées non existante dans l'image

Question 2

rMin = 1, rMax = 100, dR = 1
cMin = 1, cMax = 100, Dc=1
radMin = 5, radMax = 100*sqrt(2), dRad = 1

100*100*136 = 1 360 000 cercles possibles

Question 3

acc(1, 1, 1) : Quart de cercle de centre (1;1) avec 5 pixels de rayon
acc(10, 7, 30) : Cercle de centre (10;7) avec 35 pixels de rayon

Question 4

acc(40,40,8)

Exercice 2

Pour implémenter le détecteur de cercle, nous sommes partis de la même fonction de lissage et de détection de contour que pour le TP précédent. (flou gaussian et sobel)

Nous avons ensuite, sur ces contours, appliqué une érosion (importance de l'érosion fixée par l'utilisateur) afin d'avoir moins de pixels de contours, et donc moins de traitement.

Le seuil appliqué après le Sobel est calculé en fonction de la valeur maximale présente dans la magnitude du contour. On prend cette valeur maximale que l'on divise ensuite par 4.

Notre accumulateur était initialisé comme un tableau statique : acc[row][col][rad]. Cependant cela posait des problèmes au niveau de l'allocation (Segmentation Fault) avec l'image coins2.png à cause de sa taille. Pour pallier ce problème on a remplacé le tableau statique par des vecteurs.

Pour voter dans l'accumulateur, on parcourt chaque pixel de l'image résultant du seuil. Si un pixel est un pixel de contour, on cherche dans toutes les directions et pour tous les rayons compris entre le minimum et la maximum fixés, un pixel qui pourrait être le centre du cercle contenant ce pixel de contour. Pour chaque cercle contenant ce pixel, on va incrémenter de 1 la valeur dans l'accumulateur.

En incrémentant de 1, les votes ne sont pas répartis correctement entre les plus grands et les plus petits cercles car les grands cercles possèdent plus de pixels de contours. Nous avons dans un premier temps essayer d'incrémenter de 1/rad pour pondérer, cependant les résultats n'étaient pas concluants. Nous avons ensuite essayé plusieurs méthodes différentes mais toujours sans résultats. Donc nous sommes restés sur une incrémentation de 1 par vote.

Une fois l'accumulateur rempli, pour éviter d'avoir plusieurs cercles similaires car trop proches, lorsqu'on trouve un maximum local, on remet les voisins à 0 autour du maximum local. Par défaut le nombre de voisins remis à 0 est de 3x3x3 mais pour améliorer nos résultats ce nombre peut changer.

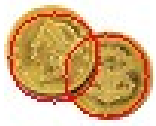
Toutes ces méthodes nous ont permis d'avoir des résultats probants, cependant nous avons été obligé pour chaque image de "personnaliser" le traitement effectué. Par exemple en changeant le nombre de voisins qui sont visités ou alors l'importance de l'érosion.

Par exemple pour l'image coins2.png :

```
#define NB_CIRCLES 8 (On prend les 8 meilleurs cercles et on les affiche)
#define SEARCH_CUBE_SIZE 10 (On regarde dans un cube 21x21x21 autour du maximum)
#define EROSION_SIZE 1 (Importance de l'érosion)

#define ROW_STEP 2
#define COL_STEP 2
#define RAD_STEP 1

#define RAD_MIN 50 (Taille des cercles recherchés en pixel)
#define RAD_MAX 100
```

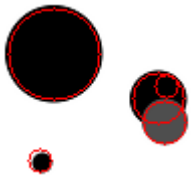
Pour l'image coins.png :

```
#define NB_CIRCLES 2
#define SEARCH_CUBE_SIZE 6
#define EROSION_SIZE 1

#define ROW_STEP 1
#define COL_STEP 1
#define RAD_STEP 1

#define RAD_MIN 15
#define RAD_MAX 30
```



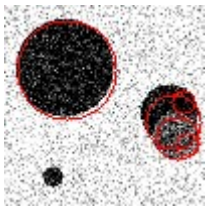
Pour l'image four.png :

```
#define NB_CIRCLES 5
#define SEARCH_CUBE_SIZE 3
#define EROSION_SIZE 1

#define ROW_STEP 1
#define COL_STEP 1
#define RAD_STEP 1

#define RAD_MIN 7
#define RAD_MAX 30
```



Pour l'image fourn.png

```
#define NB_CIRCLES 5
#define SEARCH_CUBE_SIZE 3
#define EROSION_SIZE 0

#define ROW_STEP 1
#define COL_STEP 1
#define RAD_STEP 1

#define RAD_MIN 7
#define RAD_MAX 30
```



Pour l'image MoonCoin.png :

```
#define NB_CIRCLES 2
#define SEARCH_CUBE_SIZE 1
#define EROSION_SIZE 1

#define ROW_STEP 1
#define COL_STEP 1
#define RAD_STEP 1
```

```
#define RAD_MIN 7
#define RAD_MAX 25
```

Exercice 3

Question 1

Avec notre implémentation du détecteur sans amélioration, pour four.png nous avons un temps d'exécution de : 319.414ms

On peut dire que la complexité de $O(N^4)$ car on a row*col*radius*angle. Pour une image 600x600 on aura un temps de traitement augmenté de façon exponentielle.

Question 2

Pour optimiser notre détecteur, nous avons décidé d'implémenter dans le ficher fastHough.cpp la méthode utilisant la direction du gradient pour réduire l'angle de recherche de cercles lors des votes.

Pour ce faire, on récupère l'image représentant les gradients en x et y avant le seuillage. A partir de ces gradients, on peut calculer la direction en radian (à convertir en degré) du gradient. Nous pouvons ensuite restreindre l'angle de recherche autour de cet angle (-20 à +20 degrés dans notre programme) à la place de faire une recherche à 360 degrés. La recherche est effectuée dans les deux sens (0 et +180 degrés).

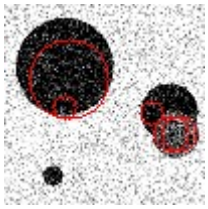
Les résultats ne sont pas aussi bons que sans l'optimisation, mais le temps de traitement est largement réduit.

Pour coins2.png (2792.2ms, précédemment 10265.1ms) :

Pour coins.png (152.662ms, précédemment 271.738ms)

Pour four.png (226.56ms, précédemment 319.414ms) :

Pour fourn.png (310.487ms, précédemment 820.599ms) :



Pour MoonCoin.png (135.068ms, précédemment 236.365ms) :